W3School Python 教程

wizardforcel

Published with GitBook



目錄

介紹 0

Python 基础教程 1

- Python 简介 1.1
- Python 环境搭建 1.2
- Python 基础语法 1.3
- Python 变量类型 1.4
- Python 运算符 1.5
- Python 条件语句 1.6
- Python 循环语句 1.7
- Python While循环语句 1.8
- Python for 循环语句 1.9
- Python 循环嵌套 1.10
- Python break 语句 1.11
- Python continue 语句 1.12
- Python pass 语句 1.13
- Python 数字 1.14
- Python 字符串 1.15
- Python 列表(Lists) 1.16
- Python 元组 1.17
- Python 字典(Dictionary) 1.18
- Python 日期和时间 1.19
- Python 函数 1.20
- Python 模块 1.21
- Python 文件I/O 1.22
- Python 异常处理 1.23

Python 高级教程 2

- Python 面向对象 2.1
- Python 正则表达式 2.2
- Python CGI编程 2.3
- Python 使用SMTP发送邮件 2.4
- Python 多线程 2.5
- Python 2.x与3??.x版本区别 2.6
- Python IDE 2.7
- Python JSON 2.8

Python3 教程 3

Python3 基础语法 3.1
Python3 基本数据类型 3.2
Python3 解释器 3.3
Python3 注释 3.4
Python3 数字运算 3.5
Python3 字符串 3.6
Python3 列表 3.7
Python3 编程第一步 3.8
Python3 条件控制 3.9
Python3 循环 3.10
Python3 函数 3.11
Python3 数据结构 3.12
Python3 模块 3.13
Python3 输入和输出 3.14
Python3 错误和异常 3.15
Python3 类 3.16
Python3 标准库概览 3.17
免责声明 4

W3School Python教程

来源：

- Python教程
- Python3教程

整理：飞龙

Python 基础教程

Python 简介

Python是一个高层次的结合了解释性、编译性、互动性和面向对象的脚本语言。

Python的设计具有很强的可读性，相比其他语言经常使用英文关键字，其他语言的一些标点符号，它具有比其他语言更有特色语法结构。

- **Python 是一种解释型语言：** 这意味着开发过程中没有了编译这个环节。类似于PHP和Perl语言。
- **Python 是交互式语言：** 这意味着，您可以在一个Python提示符，直接互动执行写你的程序。
- **Python 是面向对象语言:** 这意味着Python支持面向对象的风格或代码封装在对象的编程技术。
- **Python是初学者的语言**：Python 对初级程序员而言，是一种伟大的语言，它支持广泛的应用程序开发，从简单的文字处理到 WWW 浏览器再到游戏。

Python发展历史

Python是由Guido van Rossum在八十年代末和九十年代初，在荷兰国家数学和计算机科学研究所设计出来的。

Python 本身也是由诸多其他语言发展而来的,这包括 ABC、Modula-3、C、C++、Algol-68、SmallTalk、Unix shell 和其他的脚本语言等等。

像Perl语言一样, Python 源代码同样遵循 GPL(GNU General Public License)协议。

现在Python是由一个核心开发团队在维护，，Guido van Rossum 仍然占据着至关重要的作用，指导其进展。

Python特点

- **1.**易于学习：Python有相对较少的关键字，结构简单，和一个明确定义的语法，学习起来更加简单。
- **2.**易于阅读：Python代码定义的更清晰。
- **3.**易于维护：Python的成功在于它的源代码是相当容易维护的。
- **4.**一个广泛的标准库：Python的最大的优势之一是丰富的库，跨平台的，在UNIX，Windows和Macintosh兼容很好。
- **5.**互动模式：互动模式的支持，您可以从终端输入并获得结果的语言，互动的测试和调试代码片断。

- **6.**便携式：Python可以运行在多种硬件平台和所有平台上都具有相同的接口。
- **7.**可扩展：可以添加低层次的模块到Python解释器。这些模块使程序员可以添加或定制自己的工具，更有效。
- **8.**数据库：Python提供所有主要的商业数据库的接口。
- **9.GUI**编程：Python支持GUI可以创建和移植到许多系统调用。
- **10.**可扩展性：相比 shell 脚本，Python 提供了一个更好的结构，且支持大型程序。

Python 环境搭建

本章节我们将向大家介绍如何在本地搭建Python开发环境。

Python可应用于多平台包括 Linux 和 Mac OS X。

你可以通过终端窗口输入 "python" 命令来查看本地是否已经安装Python以及Python的安装版本。

- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, 等等。)
- Win 9x/NT/2000
- Macintosh (Intel, PPC, 68K)
- OS/2
- DOS (多个DOS版本)
- PalmOS
- Nokia 移动手机
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion
- Python 同样可以移植到 Java 和 .NET 虚拟机上。

Python下载

Python最新源码，二进制文档，新闻资讯等可以在Python的官网查看到：

Python官网：http://www.python.org/

你可以在一下链接中下载Python的文档，你可以下载 HTML、PDF 和 PostScript 等格式的文档。

Python文档下载地址：www.python.org/doc/

Python安装

Python已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。

您需要下载适用于您使用平台的二进制代码，然后安装Python。

如果您平台的二进制代码是不可用的，你需要使用C编译器手动编译源代码。

编译的源代码，功能上有更多的选择性， 为python安装提供了更多的灵活性。

以下为不同平台上安装Python的方法：

Unix & Linux 平台安装 Python:

以下为在Unix & Linux 平台上安装 Python 的简单步骤：

- 打开WEB浏览器访问http://www.python.org/download/
- 选择使用于Unix/Linux的源码压缩包。
- 下载及解压压缩包。
- 如果你需要自定义一些选项修改*Modules/Setup*
- 执行 ./configure 脚本
- make
- make install

执行以上操作后，Python会安装在 /usr/local/bin 目录中，Python库安装在/usr/local/lib/pythonXX，XX为你使用的Python的版本号。

Window 平台安装 Python:

以下为在 Window 平台上安装 Python 的简单步骤：

- 打开WEB浏览器访问http://www.python.org/download/
- 在下载列表中选择Window平台安装包，包格式为：*python-XYZ.msi* 文件，XYZ 为你要安装的版本号。
- 要使用安装程序 *python-XYZ.msi*, Windows系统必须支持Microsoft Installer 2.0搭配使用。只要保存安装文件到本地计算机，然后运行它，看看你的机器支持MSI。Windows XP和更高版本已经有MSI，很多老机器也可以安装MSI。
- 下载后，双击下载包，进入Python安装向导，安装非常简单，你只需要使用默认的设置一直点击"下一步"直到安装完成即可。

MAC 平台安装 Python:

最近的Macs系统都自带有Python环境，但是自带的Python版本为旧版本，你可以通过链接http://www.python.org/download/mac/ 查看MAC上Python的新版功能介绍。

MAC上完整的Python安装教程你可以查看：http://www.cwi.nl/~jack/macpython.html

环境变量配置

程序和可执行文件可以在许多目录，而这些路径很可能不在操作系统提供可执行文件的搜索路径中。

path(路径)存储在环境变量中，这是由操作系统维护的一个命名的字符串。这些变量包含可用的命令行解释器和其他程序的信息。

Unix或Windows中路径变量为PATH（UNIX区分大小写，Windows不区分大小写）。

在Mac OS中，安装程序过程中改变了python的安装路径。如果你需要在其他目录引用Python，你必须在path中添加Python目录。

在 Unix/Linux 设置环境变量

- 在 **csh shell:** 输入

  ```
  setenv PATH "$PATH:/usr/local/bin/python"
  ```

  , 按下"Enter"。

- 在 **bash shell (Linux):** 输入

  ```
  export PATH="$PATH:/usr/local/bin/python"
  ```

  ，按下"Enter"。

- 在 **sh** 或者 **ksh shell:** 输入

  ```
  PATH="$PATH:/usr/local/bin/python"
  ```

  , 按下"Enter"。

注意**:** /usr/local/bin/python 是Python的安装目录。

在 Windows 设置环境变量

在环境变量中添加Python目录：

- 在命令提示框中**(cmd)** **:** 输入

  ```
  path %path%;C:\Python
  ```

  , 按下"Enter"。

注意**:** C:\Python 是Python的安装目录。

Python 环境变量

下面几个重要的环境变量，它应用于Python：

变量名	描述
PYTHONPATH	PYTHONPATH是Python搜索路径，默认我们import的模块都会从PYTHONPATH里面寻找。
PYTHONSTARTUP	Python启动后，先寻找PYTHONSTARTUP环境变量，然后执行此文件中变量指定的执行代码。
PYTHONCASEOK	加入PYTHONCASEOK的环境变量, 就会使python导入模块的时候不区分大小写.
PYTHONHOME	另一种模块搜索路径。它通常内嵌于的PYTHONSTARTUP或PYTHONPATH目录中，使得两个模块库更容易切换。

运行Python

有三种方式可以运行Python：

1、交互式解释器：

你可以通过命令行窗口进入python并开在交互式解释器中开始编写Python代码。

你可以在Unix，DOS或任何其他提供了命令行或者shell的系统进行python编码工作。

$python # Unix/Linux

或者

python% # Unix/Linux

或者

C:>python # Windows/DOS

以下为Python命令行参数：

选项	描述
-d	在解析时显示调试信息
-O	生成优化代码 (.pyo 文件)
-S	启动时不引入查找Python路径的位置
-v	输出Python版本号
-X	从 1.6版本之后基于内建的异常（仅仅用于字符串）已过时。
-c cmd	执行 Python 脚本，并将运行结果作为 cmd 字符串。
file	在给定的python文件执行python脚本。

2、命令行脚本

在你的应用程序中通过引入解释器可以在命令行中执行Python脚本，如下所示：

$python script.py # Unix/Linux

或者

python% script.py # Unix/Linux

或者

C:>python script.py # Windows/DOS

注意：在执行脚本时，请检查脚本是否有可执行权限。

3、集成开发环境（IDE：Integrated Development Environment）

您可以使用图形用户界面（GUI）环境来编写及运行Python代码。以下推荐各个平台上使用的IDE：

- **Unix:** IDLE 是 UNIX 上最早的 Python IDE 。
- **Windows:** PythonWin 是一个 Python 集成开发环境,在许多方面都比 IDE 优秀
- **Macintosh:** Python 的 Mac 可以使用 IDLE IDE，你可以在网站上下载对应MAC的IDLE 。

继续下一章之前，请确保您的环境已搭建成功。如果你不能够建立正确的环境，那么你就可以从您的系统管理员的帮助。

在以后的章节中给出的例子已在Centos（Linux）下Python2.4.3版本测试通过。

Python 基础语法

Python语言与Perl，C和Java等语言有许多相似之处。但是，也存在一些差异。

在本章中我们将来学习Python的基础语法，让你快速学会Python编程。

第一个Python程序

交互式编程

交互式编程不需要创建脚本文件，是通过 Python 解释器的交互模式进来编写代码。

linux上你只需要在命令行中输入 Python 命令即可启动交互式编程,提示窗口如下：

```
$ python
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
Type "help", "copyright", "credits" or "license" for more informat:
>>>
```

Window上在安装Python时已经已经安装了默认的交互式编程客户端，提示窗口如下：

```
Python 3.3.2 Shell
File  Edit  Shell  Debug  Options  Windows  Help
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

在 python 提示符中输入以下文本信息，然后按 Enter 键查看运行效果：

```
>>> print "Hello, Python!";
```

在 Python 2.4.3 版本中,以上事例输出结果如下：

```
Hello, Python!
```

如果您运行的是新版本的Python，那么你就需要在print语句中使用括号如：

```
>>>  print ("Hello, Python!");
```

脚本式编程

通过脚本参数调用解释器开始执行脚本，直到脚本执行完毕。当脚本执行完成后，解释器不再有效。

让我们写一个简单的Python脚本程序。所有Python文件将以.py为扩展名。将以下的源代码拷贝至test.py文件中。

```
print "Hello, Python!";
```

这里，假设你已经设置了Python解释器PATH变量。使用以下命令运行程序：

```
$ python test.py
```

输出结果：

```
Hello, Python!
```

让我们尝试另一种方式来执行Python脚本。修改test.py文件，如下所示：

```
#!/usr/bin/python

print "Hello, Python!";
```

这里，假定您的Python解释器在/usr/bin目录中，使用以下命令执行脚本：

```
$ chmod +x test.py     # 脚本文件添加可执行权限
$./test.py
```

输出结果：

```
Hello, Python!
```

Python标识符

在python里，标识符有字母、数字、下划线组成。

在python中，所有标识符可以包括英文、数字以及下划线（_），但不能以数字开头。

python中的标识符是区分大小写的。

以下划线开头的标识符是有特殊意义的。以单下划线开头（_foo）的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用"from xxx import *"而导入；

以双下划线开头的（**foo**）代表类的私有成员；以双下划线开头和结尾的（foo）代表**python**里特殊方法专用的标识，如init__（）代表类的构造函数。

Python保留字符

下面的列表显示了在Python中的保留字。这些保留字不能用作常数或变数，或任何其他标识符名称。

所有Python的关键字只包含小写字母。

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

行和缩进

学习Python与其他语言最大的区别就是，Python的代码块不使用大括号（{}）来控制类，函数以及其他逻辑判断。python最具特色的就是用缩进来写模块。

缩进的空白数量是可变的，但是所有代码块语句必须包含相同的缩进空白数量，这个必须严格执行。如下所示：

```
if True:
    print "True"
else:
  print "False"
```

以下代码将会执行错误：

```
 if True:
    print "Answer"
    print "True"
else:
    print "Answer"
  print "False"
```

因此，在Python的代码块中必须使用相同数目的行首缩进空格数。

以下实例包含了相同数目的行首缩进代码语句块的例子：

```
#!/usr/bin/python

import sys

try:
  # open file stream
  file = open(file_name, "w")
except IOError:
  print "There was an error writing to", file_name
  sys.exit()
print "Enter '", file_finish,
print "' When finished"
while file_text != file_finish:
  file_text = raw_input("Enter text: ")
  if file_text == file_finish:
    # close the file
    file.close
    break
  file.write(file_text)
  file.write("\n")
file.close()
file_name = raw_input("Enter filename: ")
if len(file_name) == 0:
  print "Next time please enter something"
  sys.exit()
try:
  file = open(file_name, "r")
except IOError:
  print "There was an error reading file"
  sys.exit()
file_text = file.read()
file.close()
print file_text
```

多行语句

Python语句中一般以新行作为为语句的结束符。

但是我们可以使用斜杠（\）将一行的语句分为多行显示，如下所示：

```
total = item_one + \
       item_two + \
       item_three
```

语句中包含[], {} 或 () 括号就不需要使用多行连接符。如下实例：

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Python 引号

Python 接收单引号(')，双引号(")，三引号(''' """) 来表示字符串，引号的开始与结束必须的相同类型的。

其中三引号可以由多行组成，编写多行文本的快捷语法，常用语文档字符串，在文件的特定地点，被当做注释。

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Python注释

python中单行注释采用 # 开头。

python没有块注释，所以现在推荐的多行注释也是采用的 #比如：

```
#!/usr/bin/python

# First comment
print "Hello, Python!";  # second comment
```

输出结果：

```
Hello, Python!
```

注释可以在语句或表达式行末：

```
name = "Madisetti" # This is again comment
```

多条评论：

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Python空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。

空行与代码缩进不同，空行并不是Python语法的一部分。书写时不插入空行，Python解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

记住：空行也是程序代码的一部分。

等待用户输入

下面的程序在按回车键后就会等待用户输入：

```
#!/usr/bin/python

raw_input("\n\nPress the enter key to exit.")
```

以上代码中，"\n\n"在结果输出前会输出两个新的空行。一旦用户按下键时，程序将退出。

同一行显示多条语句

Python可以在同一行中使用多条语句，语句之间使用分号(;)分割，以下是一个简单的实例：

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

多个语句构成代码组

缩进相同的一组语句构成一个代码块，我们称之代码组。

像if、while、def和class这样的复合语句，首行以关键字开始，以冒号(:)结束，该行之后的一行或多行代码构成代码组。

我们将首行及后面的代码组称为一个子句(clause)。

如下实例：

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

命令行参数

很多程序可以执行一些操作来查看一些基本信，Python可以使用-h参数查看各参数帮助信息：

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit

[ etc. ]
```

Python 变量类型

变量存储在内存中的值。这就意味着在创建变量时会在内存中开辟一个空间。

基于变量的数据类型，解释器会分配指定内存，并决定什么数据可以被存储在内存中。

因此，变量可以指定不同的数据类型，这些变量可以存储整数，小数或字符。

变量赋值

Python中的变量不需要声明，变量的赋值操作既是变量声明和定义的过程。

每个变量在内存中创建，都包括变量的标识，名称和数据这些信息。

每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。

等号（=）用来给变量赋值。

等号（=）运算符左边是一个变量名,等号（=）运算符右边是存储在变量中的值。例如：

```
#!/usr/bin/python

counter = 100 # An integer assignment
miles = 1000.0 # A floating point
name = "John" # A string

print counter
print miles
print name
```

以上实例中，100，1000.0和"John"分别赋值给counter，miles，name变量。

执行以上程序会输出如下结果：

```
100
1000.0
John
```

多个变量赋值

Python允许你同时为多个变量赋值。例如：

```
a = b = c = 1
```

以上实例，创建一个整型对象，值为1，三个变量被分配到相同的内存空间上。

您也可以为多个对象指定多个变量。例如：

```
a, b, c = 1, 2, "john"
```

以上实例，两个整型对象1和2的分配给变量a和b，字符串对象"john"分配给变量c。

标准数据类型

在内存中存储的数据可以有多种类型。

例如，person.s年龄作为一个数值存储和他或她的地址是字母数字字符存储。

Python有一些标准类型用于定义操作上，他们和为他们每个人的存储方法可能。

Python有五个标准的数据类型：

- Numbers（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Dictionary（字典）

Python数字

数字数据类型用于存储数值。

他们是不可改变的数据类型，这意味着改变数字数据类型会分配一个新的对象。

当你指定一个值时，Number对象就会被创建：

```
var1 = 1
var2 = 10
```

您也可以使用del语句删除一些对象引用。

del语句的语法是：

```
del var1[,var2[,var3[....,varN]]]]
```

您可以通过使用del语句删除单个或多个对象。例如：

```
del var
del var_a, var_b
```

Python支持四种不同的数值类型：

- int（有符号整型）
- long（长整型[也可以代表八进制和十六进制]）
- float（浮点型）
- complex（复数）

实例

一些数值类型的实例：

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- 长整型也可以使用小写"L"，但是还是建议您使用大写"L"，避免与数字"1"混淆。Python使用"L"来显示长整型。
- Python还支持复数，复数由实数部分和虚数部分构成，可以用a + bj,或者complex(a,b)表示， 复数的实部a和虚部b都是浮点型

Python字符串

字符串或串(String)是由数字、字母、下划线组成的一串字符。

一般记为：

```
s="a1a2···an"(n>=0)
```

它是编程语言中表示文本的数据类型。

python的字串列表有2种取值顺序:

- 从左到右索引默认0开始的，最大范围是字符串长度少1
- 从右到左索引默认-1开始的，最大范围是字符串开头

如果你的实要取得一段子串的话，可以用到变量[头下标:尾下标]，就可以截取相应的字符串，其中下标是从0开始算起，可以是正数或负数，下标可以为空表示取到头或尾。

比如:

```
s = 'ilovepython'
```

s[1:5]的结果是love。

当使用以冒号分隔的字符串，python返回一个新的对象，结果包含了以这对偏移标识的连续的内容，左边的开始是包含了下边界。

上面的结果包含了s[1]的值l，而取到的最大范围不包括上边界，就是s[5]的值p。

加号（+）是字符串连接运算符，星号（*）是重复操作。如下实例：

```
#!/usr/bin/python

str = 'Hello World!'

print str # 输出完整字符串
print str[0] # 输出字符串中的第一个字符
print str[2:5] # 输出字符串中第三个至第五个之间的字符串
print str[2:] # 输出从第三个字符开始的字符串
print str * 2 # 输出字符串两次
print str + "TEST" # 输出连接的字符串
```

以上实例输出结果：

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python列表

List（列表） 是 Python 中使用最频繁的数据类型。

列表可以完成大多数集合类的数据结构实现。它支持字符，数字，字符串甚至可以包含列表（所谓嵌套）。

列表用[ ]标识。是python最通用的复合数据类型。看这段代码就明白。

列表中的值得分割也可以用到变量[头下标:尾下标]，就可以截取相应的列表，从左到右索引默认0开始的，从右到左索引默认-1开始，下标可以为空表示取到头或尾。

加号（+）是列表连接运算符，星号（*）是重复操作。如下实例：

```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list # 输出完整列表
print list[0] # 输出列表的第一个元素
print list[1:3] # 输出第二个至第三个的元素
print list[2:] # 输出从第三个开始至列表末尾的所有元素
print tinylist * 2 # 输出列表两次
print list + tinylist # 打印组合的列表
```

以上实例输出结果：

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python元组

元组是另一个数据类型，类似于List（列表）。

元组用"()"标识。内部元素用逗号隔开。但是元素不能二次赋值，相当于只读列表。

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print list # 输出完整列表
print list[0] # 输出列表的第一个元素
print list[1:3] # 输出第二个至第三个的元素
print list[2:] # 输出从第三个开始至列表末尾的所有元素
print tinylist * 2 # 输出列表两次
print list + tinylist # 打印组合的列表
```

以上实例输出结果：

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

以下是元组无效的，因为元组是不允许更新的。而列表是允许更新的：

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tuple[2] = 1000 # 元组中是非法应用
list[2] = 1000 # 列表中是合法应用
```

Python元字典

字典(dictionary)是除列表意外python之中最灵活的内置数据结构类型。列表是有序的对象结合，字典是无序的对象集合。

两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。

字典用"{ }"标识。字典由索引(key)和它对应的值value组成。

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print dict['one'] # 输出键为'one' 的值
print dict[2] # 输出键为 2 的值
print tinydict # 输出完整的字典
print tinydict.keys() # 输出所有键
print tinydict.values() # 输出所有值
```

输出结果为：

```
This is one This is two {'dept': 'sales', 'code': 6734, 'name': 'jo
```

Python数据类型转换

有时候，我们需要对数据内置的类型进行转换，数据类型的转换，你只需要将数据类型作为函数名即可。

以下几个内置的函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。

函数	描述
int(x [,base])	将x转换为一个整数
long(x [,base])	将x转换为一个长整数
float(x)	将x转换到一个浮点数
complex(real [,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串
eval(str)	用来计算在字符串中的有效Python表达式,并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
set(s)	转换为可变集合
dict(d)	创建一个字典。d 必须是一个序列 (key,value)元组。
frozenset(s)	转换为不可变集合
chr(x)	将一个整数转换为一个字符
unichr(x)	将一个整数转换为Unicode字符
ord(x)	将一个字符转换为它的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

Python 运算符

什么是运算符？

本章节主要说明Python的运算符。举个简单的例子 **4 +5 = 9** 。 例子中，4和5被称为操作数，"+"号为运算符。

Python语言支持以下类型的运算符:

- 算术运算符
- 比较（关系）运算符
- 赋值运算符
- 逻辑运算符
- 位运算符
- 成员运算符
- 身份运算符
- 运算符优先级

接下来让我们一个个来学习Python的运算符。

Python算术运算符

以下假设变量a为10，变量b为20：

运算符	描述	实例
+	加 - 两个对象相加	a + b 输出结果 30
-	减 - 得到负数或是一个数减去另一个数	a - b 输出结果 -10
*	乘 - 两个数相乘或是返回一个被重复若干次的字符串	a * b 输出结果 200
/	除 - x除以y	b / a 输出结果 2
%	取模 - 返回除法的余数	b % a 输出结果 0
**	幂 - 返回x的y次幂	a**b 为10的20次方， 输出结果 100000000000000000000
//	取整除 - 返回商的整数部分	9//2 输出结果 4 , 9.0//2.0 输出结果 4.0

以下实例演示了Python所有算术运算符的操作：

```
#!/usr/bin/python

a = 21
b = 10
c = 0

c = a + b
print "Line 1 - Value of c is ", c

c = a - b
print "Line 2 - Value of c is ", c

c = a * b
print "Line 3 - Value of c is ", c

c = a / b
print "Line 4 - Value of c is ", c

c = a % b
print "Line 5 - Value of c is ", c

a = 2
b = 3
c = a**b
print "Line 6 - Value of c is ", c

a = 10
b = 5
c = a//b
print "Line 7 - Value of c is ", c
```

以上实例输出结果：

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 8
Line 7 - Value of c is 2
```

Python比较运算符

以下假设变量a为10，变量b为20：

运算符	描述	实例
==	等于 - 比较对象是否相等	(a == b) 返回 False。
!=	不等于 - 比较两个对象是否不相等	(a != b) 返回 true.
<>	不等于 - 比较两个对象是否不相等	(a <> b) 返回 true。这个运算符类似 != 。
>	大于 - 返回x是否大于y	(a > b) 返回 False。
<	小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。	(a < b) 返回 true。
>=	大于等于 - 返回x是否大于等于y。	(a >= b) 返回 False。
<=	小于等于 - 返回x是否小于等于y。	(a <= b) 返回 true。

以下实例演示了Python所有比较运算符的操作：

```
#!/usr/bin/python

a = 21
b = 10
c = 0

if ( a == b ):
   print "Line 1 - a is equal to b"
else:
   print "Line 1 - a is not equal to b"

if ( a != b ):
   print "Line 2 - a is not equal to b"
else:
   print "Line 2 - a is equal to b"

if ( a <> b ):
   print "Line 3 - a is not equal to b"
else:
   print "Line 3 - a is equal to b"

if ( a < b ):
   print "Line 4 - a is less than b"
else:
   print "Line 4 - a is not less than b"

if ( a > b ):
   print "Line 5 - a is greater than b"
else:
   print "Line 5 - a is not greater than b"

a = 5;
b = 20;
if ( a <= b ):
   print "Line 6 - a is either less than or equal to  b"
else:
   print "Line 6 - a is neither less than nor equal to  b"

if ( b >= a ):
   print "Line 7 - b is either greater than  or equal to b"
else:
   print "Line 7 - b is neither greater than  nor equal to b"
```

以上实例输出结果：

```
Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not equal to b
Line 4 - a is not less than b
Line 5 - a is greater than b
Line 6 - a is either less than or equal to b
Line 7 - b is either greater than or equal to b
```

Python赋值运算符

以下假设变量a为10，变量b为20：

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c = *a* 等效于 *c* = *c* a
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a** 等效于 **c = c** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

以下实例演示了Python所有赋值运算符的操作：

```
#!/usr/bin/python

a = 21
b = 10
c = 0

c = a + b
print "Line 1 - Value of c is ", c

c += a
print "Line 2 - Value of c is ", c

c *= a
print "Line 3 - Value of c is ", c

c /= a
print "Line 4 - Value of c is ", c

c  = 2
c %= a
print "Line 5 - Value of c is ", c

c **= a
print "Line 6 - Value of c is ", c

c //= a
print "Line 7 - Value of c is ", c
```

以上实例输出结果：

```
Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52
Line 5 - Value of c is 2
Line 6 - Value of c is 2097152
Line 7 - Value of c is 99864
```

Python位运算符

按位运算符是把数字看作二进制来进行计算的。Python中的按位运算法则如下：

运算符	描述	实例
&	按位与运算符	(a & b) 输出结果 12 ，二进制解释： 0000 1100
|	按位或运算符	(a | b) 输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符	(~a) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。
<<	左移动运算符	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符	a >> 2 输出结果 15 ，二进制解释： 0000 1111

以下实例演示了Python所有位运算符的操作：

```
#!/usr/bin/python

a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0

c = a & b;        # 12 = 0000 1100
print "Line 1 - Value of c is ", c

c = a | b;        # 61 = 0011 1101
print "Line 2 - Value of c is ", c

c = a ^ b;        # 49 = 0011 0001
print "Line 3 - Value of c is ", c

c = ~a;           # -61 = 1100 0011
print "Line 4 - Value of c is ", c

c = a << 2;       # 240 = 1111 0000
print "Line 5 - Value of c is ", c

c = a >> 2;       # 15 = 0000 1111
print "Line 6 - Value of c is ", c
```

以上实例输出结果：

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Python逻辑运算符

Python语言支持逻辑运算符，以下假设变量a为10，变量b为20：

运算符	描述	实例
and	布尔"与" - 如果x为False，x and y返回False，否则它返回y的计算值。	(a and b) 返回 true。
or	布尔"或" - 如果x是True，它返回True，否则它返回y的计算值。	(a or b) 返回 true。
not	布尔"非" - 如果x为True，返回False。如果x为False，它返回True。	not(a and b) 返回 false。

以下实例演示了Python所有逻辑运算符的操作：

```
#!/usr/bin/python

a = 10
b = 20
c = 0

if ( a and b ):
   print "Line 1 - a and b are true"
else:
   print "Line 1 - Either a is not true or b is not true"

if ( a or b ):
   print "Line 2 - Either a is true or b is true or both are true"
else:
   print "Line 2 - Neither a is true nor b is true"

a = 0
if ( a and b ):
   print "Line 3 - a and b are true"
else:
   print "Line 3 - Either a is not true or b is not true"

if ( a or b ):
   print "Line 4 - Either a is true or b is true or both are true"
else:
   print "Line 4 - Neither a is true nor b is true"

if not( a and b ):
   print "Line 5 - a and b are true"
else:
   print "Line 5 - Either a is not true or b is not true"
```

以上实例输出结果：

```
Line 1 - a and b are true
Line 2 - Either a is true or b is true or both are true
Line 3 - Either a is not true or b is not true
Line 4 - Either a is true or b is true or both are true
Line 5 - a and b are true
```

Python成员运算符

除了以上的一些运算符之外，Python还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。

运算符	描述	实例
in	如果在指定的序列中找到值返回True，否则返回False。	x 在 y序列中 , 如果x在y序列中返回True。
not in	如果在指定的序列中没有找到值返回True，否则返回False。	x 不在 y序列中 , 如果x不在y序列中返回True。

以下实例演示了Python所有成员运算符的操作：

```
#!/usr/bin/python

a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
   print "Line 1 - a is available in the given list"
else:
   print "Line 1 - a is not available in the given list"

if ( b not in list ):
   print "Line 2 - b is not available in the given list"
else:
   print "Line 2 - b is available in the given list"

a = 2
if ( a in list ):
   print "Line 3 - a is available in the given list"
else:
   print "Line 3 - a is not available in the given list"
```

以上实例输出结果：

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

Python身份运算符

身份运算符用于比较两个对象的存储单元

运算符	描述	实例
is	is是判断两个标识符是不是引用自一个对象	x is y, 如果 id(x) 等于 id(y) , **is** 返回结果 1
is not	is not是判断两个标识符是不是引用自不同对象	x is not y, 如果 id(x) 不等于 id(y). **is not** 返回结果 1

以下实例演示了Python所有身份运算符的操作：

```
#!/usr/bin/python

a = 20
b = 20

if ( a is b ):
   print "Line 1 - a and b have same identity"
else:
   print "Line 1 - a and b do not have same identity"

if ( id(a) == id(b) ):
   print "Line 2 - a and b have same identity"
else:
   print "Line 2 - a and b do not have same identity"

b = 30
if ( a is b ):
   print "Line 3 - a and b have same identity"
else:
   print "Line 3 - a and b do not have same identity"

if ( a is not b ):
   print "Line 4 - a and b do not have same identity"
else:
   print "Line 4 - a and b have same identity"
```

以上实例输出结果：

```
Line 1 - a and b have same identity
Line 2 - a and b have same identity
Line 3 - a and b do not have same identity
Line 4 - a and b do not have same identity
```

Python运算符优先级

以下表格列出了从最高到最低优先级的所有运算符：

运算符	描述
**	指数 (最高优先级)
~ + -	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
* / % //	乘，除，取模和取整除
+ -	加法减法
>> <<	右移，左移运算符
&	位 'AND'
^ |	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += = *=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not or and	逻辑运算符

以下实例演示了Python所有运算符优先级的操作：

```
#!/usr/bin/python

a = 20
b = 10
c = 15
d = 5
e = 0

e = (a + b) * c / d       #( 30 * 15 ) / 5
print "Value of (a + b) * c / d is ",  e

e = ((a + b) * c) / d     # (30 * 15 ) / 5
print "Value of ((a + b) * c) / d is ",  e

e = (a + b) * (c / d);    # (30) * (15/5)
print "Value of (a + b) * (c / d) is ",  e

e = a + (b * c) / d;      #  20 + (150/5)
print "Value of a + (b * c) / d is ",  e
```

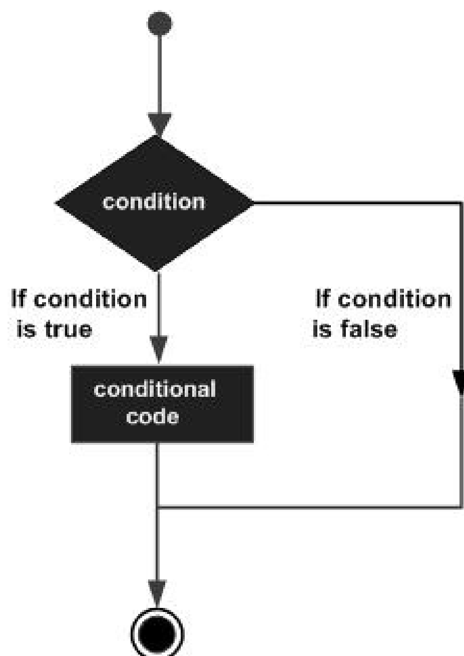
以上实例输出结果：

```
Value of (a + b) * c / d is 90
Value of ((a + b) * c) / d is 90
Value of (a + b) * (c / d) is 90
Value of a + (b * c) / d is 50
```

Python 条件语句

Python条件语句是通过一条或多条语句的执行结果（True或者False）来决定执行的代码块。

可以通过下图来简单了解条件语句的执行过程:



Python程序语言指定任何非0和非空（null）值为true，0 或者 null为false。

Python 编程中 if 语句用于控制程序的执行，基本形式为：

```
if 判断条件：
    执行语句……
else：
    执行语句……
```

其中"判断条件"成立时（非零），则执行后面的语句，而执行内容可以多行，以缩进来区分表示同一范围。

else 为可选语句，当需要在条件不成立时执行内容则可以执行相关语句，具体例子如下：

```
# coding=utf8
# 例1：if 基本用法

flag = False
name = 'luren'
if name == 'python':          # 判断变量否为'python'
    flag = True          # 条件成立时设置标志为真
    print 'welcome boss'    # 并输出欢迎信息
else:
    print name              # 条件不成立时输出变量名称
```

输出结果为：

```
>>> luren             # 输出结果
```

if 语句的判断条件可以用>（大于）、<(小于)、==（等于）、>=（大于等于）、<=（小于等于）来表示其关系。

当判断条件为多个值是，可以使用以下形式：

```
if 判断条件1:
    执行语句1……
elif 判断条件2:
    执行语句2……
elif 判断条件3:
    执行语句3……
else:
    执行语句4……
```

实例如下：

```
# coding=utf8
# 例2：elif用法

num = 5
if num == 3:            # 判断num的值
    print 'boss'
elif num == 2:
    print 'user'
elif num == 1:
    print 'worker'
elif num < 0:           # 值小于零时输出
    print 'error'
else:
    print 'roadman'     # 条件均不成立时输出
```

输出结果为：

```
>>> roadman         # 输出结果
```

由于 python 并不支持 switch 语句，所以多个条件判断，只能用 elif 来实现，如果判断需要多个条件需同时判断时，可以使用 or （或），表示两个条件有一个成立时判断条件成功；使用 and （与）时，表示只有两个条件同时成立的情况下，判断条件才成功。

```
# coding=utf8
# 例3：if语句多个条件

num = 9
if num >= 0 and num <= 10:    # 判断值是否在0~10之间
    print 'hello'
>>> hello         # 输出结果

num = 10
if num < 0 or num > 10:    # 判断值是否在小于0或大于10
    print 'hello'
else:
    print 'undefine'
>>> undefine        # 输出结果

num = 8
# 判断值是否在0~5或者10~15之间
if (num >= 0 and num <= 5) or (num >= 10 and num <= 15):
    print 'hello'
else:
    print 'undefine'
>>> undefine        # 输出结果
```

当if有多个条件时可使用括号来区分判断的先后顺序，括号中的判断优先执行，此外 and 和 or 的优先级低于>（大于）、<（小于）等判断符号，即大于和小于在没有括号的情况下会比与或要优先判断。

简单的语句组

你也可以在同一行的位置上使用if条件判断语句，如下实例：

```
#!/usr/bin/python

var = 100

if ( var  == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

以上代码执行输出结果如下：

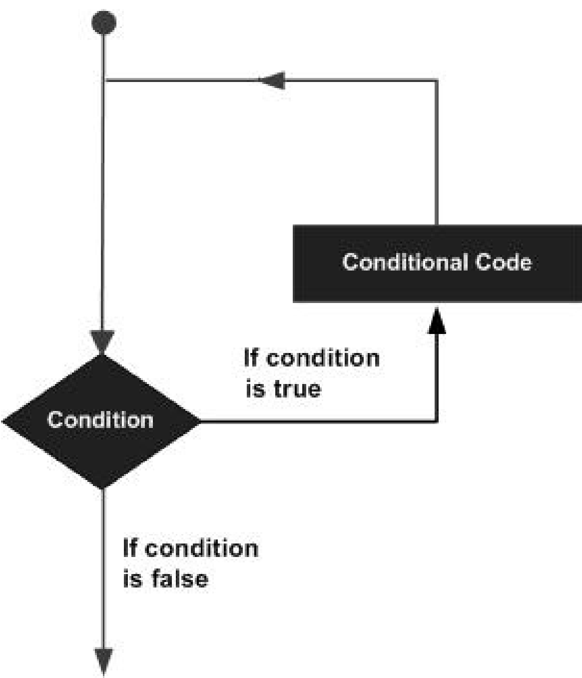
```
Value of expression is 100
Good bye!
```

Python 循环语句

本章节将向大家介绍Python的循环语句，程序在一般情况下是按顺序执行的。

编程语言提供了各种控制结构，允许更复杂的执行路径。

循环语句允许我们执行一个语句或语句组多次，下面是在大多数编程语言中的循环语句的一般形式：



Python提供了for循环和while循环（在Python中没有do..while循环）：

循环类型	描述
while 循环	在给定的判断条件为 true 时执行循环体，否则退出循环体。
for 循环	重复执行语句
嵌套循环	你可以在while循环体中嵌套for循环

循环控制语句

循环控制语句可以更改语句执行的顺序。Python支持以下循环控制语句：

控制语句	描述
break 语句	在语句块执行过程中终止循环，并且跳出整个循环
continue 语句	在语句块执行过程中终止当前循环，跳出该次循环，执行下一次循环。
pass 语句	pass是空语句，是为了保持程序结构的完整性。

Python While循环语句

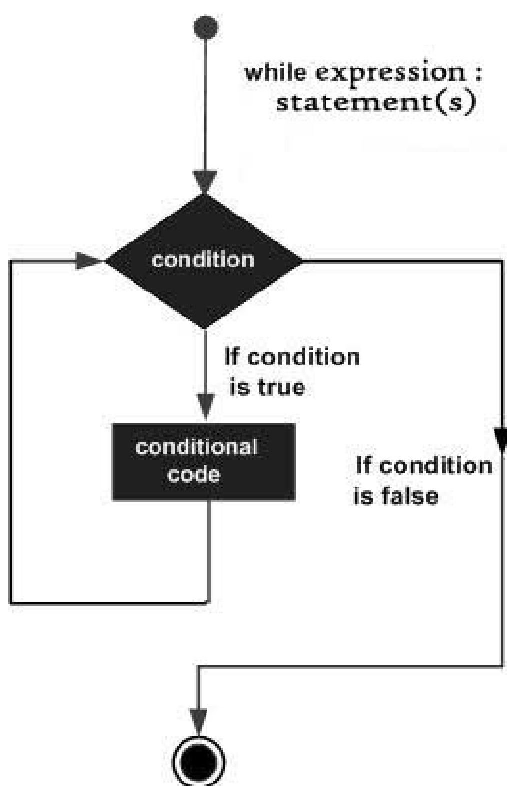
Python 编程中 while 语句用于循环执行程序，即在某条件下，循环执行某段程序，以处理需要重复处理的相同任务。其基本形式为：

```
while 判断条件：
    执行语句……
```

执行语句可以是单个语句或语句块。判断条件可以是任何表达式，任何非零、或非空（null）的值均为true。

当判断条件假false时，循环结束。

执行流程图如下：



实例：

```
#!/usr/bin/python

count = 0
while (count < 9):
   print 'The count is:', count
   count = count + 1

print "Good bye!"
```

以上代码执行输出结果:

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

while 语句时还有另外两个重要的命令 continue，break 来跳过循环，continue 用于跳过该次循环，break 则是用于退出循环，此外"判断条件"还可以是个常值，表示循环必定成立，具体用法如下：

```
# continue 和 break 用法

i = 1
while i < 10:
    i += 1
    if i%2 > 0:     # 非双数时跳过输出
        continue
    print i         # 输出双数2、4、6、8、10

i = 1
while 1:            # 循环条件为1必定成立
    print i         # 输出1~10
    i += 1
    if i > 10:     # 当i大于10时跳出循环
        break
```

无限循环

如果条件判断语句永远为 true，循环将会无限的执行下去，如下实例：

```
#!/usr/bin/python

var = 1
while var == 1 :  # 该条件永远为true，循环将无限执行下去
   num = raw_input("Enter a number  :")
   print "You entered: ", num

print "Good bye!"
```

以上实例输出结果：

```
Enter a number  :20
You entered:  20
Enter a number  :29
You entered:  29
Enter a number  :3
You entered:  3
Enter a number between :Traceback (most recent call last):
  File "test.py", line 5, in <module>
    num = raw_input("Enter a number :")
KeyboardInterrupt
```

注意：以上的无限循环你可以使用 CTRL+C 来中断循环。

循环使用 else 语句

在 python 中，for … else 表示这样的意思，for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完（即 for 不是通过 break 跳出而中断的）的情况下执行，while … else 也是一样。

```
#!/usr/bin/python

count = 0
while count < 5:
   print count, " is  less than 5"
   count = count + 1
else:
   print count, " is not less than 5"
```

以上实例输出结果为：

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

简单语句组

类似if语句的语法，如果你的while循环体中只有一条语句，你可以将该语句与while写在同一行中， 如下所示：

```
#!/usr/bin/python

flag = 1

while (flag): print 'Given flag is really true!'

print "Good bye!"
```

注意：以上的无限循环你可以使用 CTRL+C 来中断循环。

Python for 循环语句

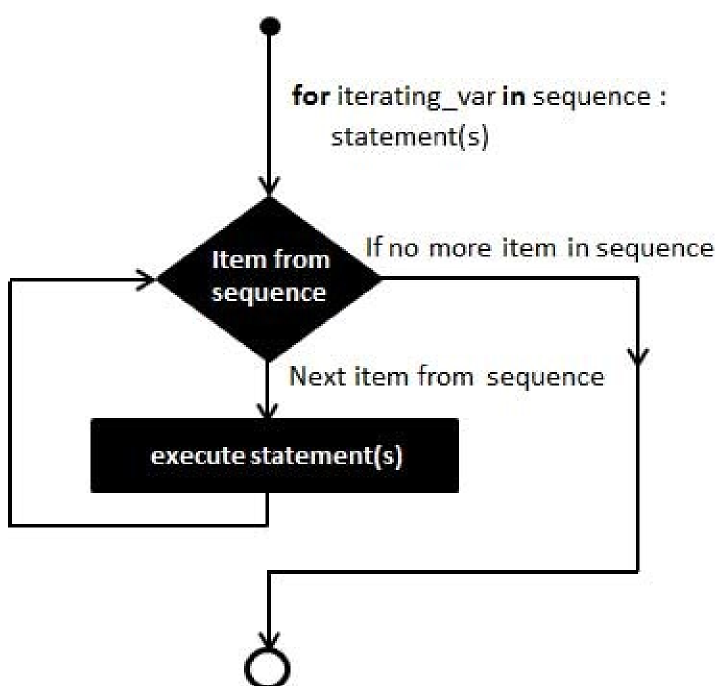
Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

语法：

for循环的语法格式如下：

```
for iterating_var in sequence:
   statements(s)
```

流程图：



实例：

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   print 'Current Letter :', letter

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print 'Current fruit :', fruit

print "Good bye!"
```

以上实例输出结果:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

通过序列索引迭代

另外一种执行循环的遍历方式是通过索引，如下实例：

```
#!/usr/bin/python

fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print 'Current fruit :', fruits[index]

print "Good bye!"
```

以上实例输出结果：

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

以上实例我们使用了内置函数 len() 和 range(),函数 len() 返回列表的长度，即元素的个数。 range返回一个序列的数。

循环使用 else 语句

在 python 中，for … else 表示这样的意思，for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完（即 for 不是通过 break 跳出而中断的）的情况下执行，while … else 也是一样。

如下实例：

```
#!/usr/bin/python

for num in range(10,20):  #to iterate between 10 to 20
   for i in range(2,num): #to iterate on the factors of the number
      if num%i == 0:      #to determine the first factor
         j=num/i          #to calculate the second factor
         print '%d equals %d * %d' % (num,i,j)
         break #to move to the next number, the #first FOR
   else:                  # else part of the loop
      print num, 'is a prime number'
```

以上实例输出结果：

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

Python 循环嵌套

Python 语言允许在一个循环体里面嵌入另一个循环。

Python for 循环嵌套语法：

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

Python while 循环嵌套语法：

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

你可以在循环体内嵌入其他的循环体，如在while循环中可以嵌入for循环， 反之，你可以在for循环中嵌入while循环。

实例：

以下实例使用了嵌套循环输出2~100之间的素数：

```
#!/usr/bin/python

i = 2
while(i < 100):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
      j = j + 1
   if (j > i/j) : print i, " 是素数"
   i = i + 1

print "Good bye!"
```

以上实例输出结果:

```
2 是素数
3 是素数
5 是素数
7 是素数
11 是素数
13 是素数
17 是素数
19 是素数
23 是素数
29 是素数
31 是素数
37 是素数
41 是素数
43 是素数
47 是素数
53 是素数
59 是素数
61 是素数
67 是素数
71 是素数
73 是素数
79 是素数
83 是素数
89 是素数
97 是素数
Good bye!
```

Python break 语句

Python break语句，就像在C语言中，打破了最小封闭for或while循环。

break语句用来终止循环语句，即循环条件没有False条件或者序列还没被完全递归完，也会停止执行循环语句。

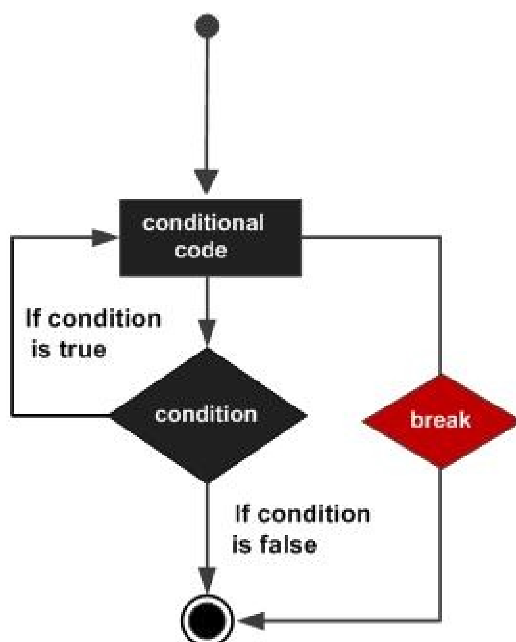
break语句用在while和for循环中。

如果您使用嵌套循环，break语句将停止执行最深层的循环，并开始执行下一行代码。

Python语言 break 语句语法：

```
break
```

流程图：



实例：

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
   print 'Current variable value :', var
   var = var -1
   if var == 5:
      break

print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

Python continue 语句

Python continue 语句跳出本次循环，而break跳出整个循环。

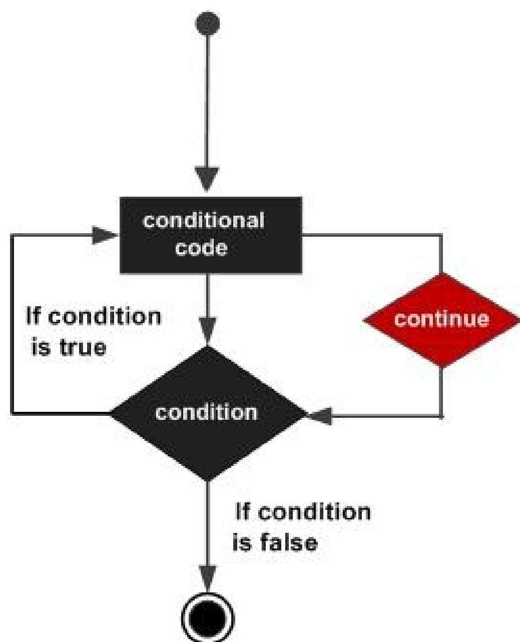
continue 语句用来告诉Python跳过当前循环的剩余语句，然后继续进行下一轮循环。

continue语句用在while和for循环中。

Python 语言 continue 语句语法格式如下：

```
continue
```

流程图：



实例：

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   if letter == 'h':
      continue
   print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print 'Current variable value :', var
print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

Python pass 语句

Python pass是空语句，是为了保持程序结构的完整性。

Python 语言 pass 语句语法格式如下：

```
pass
```

实例：

```
#!/usr/bin/python

for letter in 'Python':
   if letter == 'h':
      pass
      print 'This is pass block'
   print 'Current Letter :', letter

print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

Python 数字

Python 数字数据类型用于存储数值。

数据类型是不允许改变的,这就意味着如果改变数字数据类型得值，将重新分配内存空间。

以下实例在变量赋值时数字对象将被创建：

```
var1 = 1
var2 = 10
```

您也可以使用del语句删除一些数字对象引用。

del语句的语法是：

```
del var1[,var2[,var3[....,varN]]]]
```

您可以通过使用del语句删除单个或多个对象，例如：

```
del var
del var_a, var_b
```

Python 支持四种不同的数值类型：

- **整型(Int)** - 通常被称为是整型或整数，是正或负整数，不带小数点。
- **长整型(long integers)** - 无限大小的整数，整数最后是一个大写或小写的L。
- **浮点型(floating point real values)** - 浮点型由整数部分与小数部分组成，浮点型也可以使用科学计数法表示（2.5e2 = 2.5 x 10^2 = 250）
- **复数((complex numbers))** - 复数的虚部以字母J 或 j结尾 。如：2+3i

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- 长整型也可以使用小写"L"，但是还是建议您使用大写"L"，避免与数字"1"混淆。Python使用"L"来显示长整型。
- Python还支持复数，复数由实数部分和虚数部分构成，可以用a + bj,或者complex(a,b)表示， 复数的实部a和虚部b都是浮点型

Python数字类型转换

```
int(x [,base ])         将x转换为一个整数
long(x [,base ])        将x转换为一个长整数
float(x )               将x转换到一个浮点数
complex(real [,imag ])  创建一个复数
str(x )                 将对象 x 转换为字符串
repr(x )                将对象 x 转换为表达式字符串
eval(str )              用来计算在字符串中的有效Python表达式,并返回一个对
tuple(s )               将序列 s 转换为一个元组
list(s )                将序列 s 转换为一个列表
chr(x )                 将一个整数转换为一个字符
unichr(x )              将一个整数转换为Unicode字符
ord(x )                 将一个字符转换为它的整数值
hex(x )                 将一个整数转换为一个十六进制字符串
oct(x )                 将一个整数转换为一个八进制字符串
```

Python数学函数

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如abs(-10) 返回 10
ceil(x)	返回数字的上入整数，如math.ceil(4.1) 返回 5
cmp(x, y)	如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1
exp(x)	返回e的x次幂(e^x),如math.exp(1) 返回2.718281828459045
fabs(x)	返回数字的绝对值，如math.fabs(-10) 返回10.0
floor(x)	返回数字的下舍整数，如math.floor(4.9)返回 4
log(x)	如math.log(math.e)返回1.0,math.log(100,10)返回2.0
log10(x)	返回以10为基数的x的对数，如math.log10(100)返回 2.0
max(x1, x2,...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)	返回给定参数的最小值，参数可以为序列。
modf(x)	返回x的整数部分与小数部分，两部分的数值符号与x相同，整数部分以浮点型表示。
pow(x, y)	x**y 运算后的值。
round(x [,n])	返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。
sqrt(x)	返回数字x的平方根，数字可以为负数，返回类型为实数，如math.sqrt(4)返回 2+0j

Python随机数函数

随机数可以用于数学，游戏，安全等领域中，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。

Python包含以下常用随机数函数：

函数	描述
choice(seq)	从序列的元素中随机挑选一个元素，比如random.choice(range(10))，从0到9中随机挑选一个整数。
randrange ([start,] stop [,step])	从指定范围内，按指定基数递增的集合中获取一个随机数，基数缺省值为1
random()	随机生成下一个实数，它在[0,1)范围内。
seed([x])	改变随机数生成器的种子seed。如果你不了解其原理，你不必特别去设定seed，Python会帮你选择seed。
shuffle(lst)	将序列的所有元素随机排序
uniform(x, y)	随机生成下一个实数，它在[x,y]范围内。

Python三角函数

Python包括以下三角函数：

函数	描述
acos(x)	返回x的反余弦弧度值。
asin(x)	返回x的反正弦弧度值。
atan(x)	返回x的反正切弧度值。
atan2(y, x)	返回给定的 X 及 Y 坐标值的反正切值。
cos(x)	返回x的弧度的余弦值。
hypot(x, y)	返回欧几里德范数 sqrt(x*x* + *y*y)。
sin(x)	返回的x弧度的正弦值。
tan(x)	返回x弧度的正切值。
degrees(x)	将弧度转换为角度,如degrees(math.pi/2)， 返回90.0
radians(x)	将角度转换为弧度

Python数学常量

常量	描述
pi	数学常量 pi（圆周率，一般以π来表示）
e	数学常量 e，e即自然常数（自然常数）。

Python 字符串

字符串是最 Python 总常用的数据类型。我们可以使用引号来创建字符串。

创建字符串很简单，只要为变量分配一个值即可。例如：

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Python访问字符串中的值

Python不支持单字符类型，单字符也在Python也是作为一个字符串使用。

Python访问子字符串，可以使用方括号来截取字符串，如下实例：

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

以上实例执行结果：

```
var1[0]:  H
var2[1:5]:  ytho
```

Python字符串更新

你可以对已存在的字符串进行修改，并赋值给另一个变量，如下实例：

```
#!/usr/bin/python

var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'
```

以上实例执行结果

```
Updated String :-  Hello Python
```

Python转义字符

在需要在字符中使用特殊字符时，python用反斜杠()转义字符。如下表：

转义字符	描述
(在行尾时)	续行符
\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格(Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数，yy代表的字符，例如：\o12代表换行
\xyy	十六进制数，yy代表的字符，例如：\x0a代表换行
\other	其它的字符以普通格式输出

Python字符串运算符

下表实例变量a值为字符串"Hello"，b变量值为"Python"：

操作符	描述	实例
+	字符串连接	a + b 输出结果：HelloPython
*	重复输出字符串	a*2 输出结果：HelloHello
[]	通过索引获取字符串中字符	a[1] 输出结果 **e**
[:]	截取字符串中的一部分	a[1:4] 输出结果 **ell**
in	成员运算符 - 如果字符串中包含给定的字符返回 True	**H in a** 输出结果 1
not in	成员运算符 - 如果字符串中不包含给定的字符返回 True	**M not in a** 输出结果 1
r/R	原始字符串 - 原始字符串：所有的字符串都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。 原始字符串除在字符串的第一个引号前加上字母"r"（可以大小写）以外，与普通字符串有着几乎完全相同的语法。	**print r'\n'** prints \n 和 **print R'\n'** prints \n
%	格式字符串	情看一下章节

Python字符串格式化

Python 支持格式化字符串的输出 。尽管这样可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 %s 的字符串中。

在 Python 中，字符串格式化使用与 C 中 sprintf 函数一样的语法。

如下实例：

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

以上实例输出结果：

```
My name is Zara and weight is 21 kg!
```

python字符串格式化符号:

符 号	描述
%c	格式化字符及其ASCII码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f和%e的简写
%G	%f 和 %E 的简写
%p	用十六进制数格式化变量的地址

格式化操作符辅助指令:

符号	功能
*	定义宽度或者小数点精度
-	用做左对齐
+	在正数前面显示加号(+)
<sp>	在正数前面显示空格
#	在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X')
0	显示的数字前面填充'0'而不是默认的空格
%	'%%'输出一个单一的'%'
(var)	映射变量(字典参数)
m.n.	m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话)

Python三引号（triple quotes）

python中三引号可以将复杂的字符串进行复制:

python三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。

三引号的语法是一对连续的单引号或者双引号（通常都是成对的用）。

```
 >>> hi = '''hi
there'''
>>> hi   # repr()
'hi\nthere'
>>> print hi  # str()
hi
there
```

三引号让程序员从引号和特殊字符串的泥潭里面解脱出来，自始至终保持一小块字符串的格式是所谓的WYSIWYG（所见即所得）格式的。

一个典型的用例是，当你需要一块HTML或者SQL时，这时用字符串组合，特殊字符串转义将会非常的繁琐。

```
 errHTML = '''
<HTML><HEAD><TITLE>
Friends CGI Demo</TITLE></HEAD>
<BODY><H3>ERROR</H3>
<B>%s</B><P>
<FORM><INPUT TYPE=button VALUE=Back
ONCLICK="window.history.back()"></FORM>
</BODY></HTML>
'''
cursor.execute('''
CREATE TABLE users (
login VARCHAR(8),
uid INTEGER,
prid INTEGER)
''')
```

Unicode 字符串

Python 中定义一个 Unicode 字符串和定义一个普通字符串一样简单：

```
>>> u'Hello World !'
u'Hello World !'
```

引号前小写的"u"表示这里创建的是一个 Unicode 字符串。如果你想加入一个特殊字符，可以使用 Python 的 Unicode-Escape 编码。如下例所示：

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

被替换的 \u0020 标识表示在给定位置插入编码值为 0x0020 的 Unicode 字符（空格符）。

python的字符串内建函数

字符串方法是从python1.6到2.0慢慢加进来的——它们也被加到了Jython中。

这些方法实现了string模块的大部分方法，如下表所示列出了目前字符串内建支持的方法，所有的方法都包含了对Unicode的支持，有一些甚至是专门用于Unicode的。

方法	描述
string.capitalize()	把字符串的第一个字符大写
string.center(width)	返回一个原字符串居中,并使用空格填充至长度 width 的新字符串
string.count(str, beg=0, end=len(string))	返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数
string.decode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式解码 string，如果出错默认报一个 ValueError 的 异 常 ， 除 非 errors 指 定 的 是 'ignore' 或 者'replace'
string.encode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式编码 string，如果出错默认报一个 ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
string.endswith(obj, beg=0, end=len(string))	检查字符串是否以 obj 结束，如果 beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False.
string.expandtabs(tabsize=8)	把字符串 string 中的 tab 符号转为空格，默认的空格数 tabsize 是 8.
string.find(str, beg=0, end=len(string))	检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1
string.index(str, beg=0,	跟find()方法一样，只不过如果str不在

end=len(string))	string中会报一个异常.
string.isalnum()	如果 string 至少有一个字符并且所有字符都是字母或数字则返 回 True,否则返回 False
string.isalpha()	如果 string 至少有一个字符并且所有字符都是字母则返回 True, 否则返回 False
string.isdecimal()	如果 string 只包含十进制数字则返回 True 否则返回 False.
string.isdigit()	如果 string 只包含数字则返回 True 否则返回 False.
string.islower()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False
string.isnumeric()	如果 string 中只包含数字字符，则返回 True，否则返回 False
string.isspace()	如果 string 中只包含空格，则返回 True，否则返回 False.
string.istitle()	如果 string 是标题化的(见 title())则返回 True，否则返回 False
string.isupper()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
string.join(seq)	Merges (concatenates)以 string 作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
string.ljust(width)	返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串
string.lower()	转换 string 中所有大写字符为小写.
string.lstrip()	截掉 string 左边的空格
string.maketrans(intab, outtab])	maketrans() 方法用于创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。
max(str)	返回字符串 *str* 中最大的字母。

min(str)	返回字符串 *str* 中最小的字母。
string.partition(str)	有点像 find()和 split()的结合体,从 str 出现的第一个位置起,把 字 符 串 string 分 成 一 个 3 元 素 的 元 组 (string_pre_str,str,string_post_str),如果 string 中不包含str 则 string_pre_str == string.
string.replace(str1, str2, num=string.count(str1))	把 string 中的 str1 替换成 str2,如果 num 指定，则替换不超过 num 次.
string.rfind(str, beg=0,end=len(string))	类似于 find()函数，不过是从右边开始查找.
string.rindex(str, beg=0,end=len(string))	类似于 index()，不过是从右边开始.
string.rjust(width)	返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串
string.rpartition(str)	类似于 partition()函数,不过是从右边开始查找.
string.rstrip()	删除 string 字符串末尾的空格.
string.split(str="", num=string.count(str))	以 str 为分隔符切片 string，如果 num有指定值，则仅分隔 num 个子字符串
string.splitlines(num=string.count('\n'))	按照行分隔，返回一个包含各行作为元素的列表，如果 num 指定则仅切片 num 个行.
string.startswith(obj, beg=0,end=len(string))	检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果 beg 和 end 指定值，则在指定范围内检查.
string.strip([obj])	在 string 上执行 lstrip()和 rstrip()
string.swapcase()	翻转 string 中的大小写
string.title()	返回"标题化"的 string,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())
string.translate(str, del="")	根据 str 给出的表(包含 256 个字符)转换 string 的字符, 要过滤掉的字符放到 del 参数中
string.upper()	转换 string 中的小写字母为大写
string.zfill(width)	返回长度为 width 的字符串，原字符

string.zfill(width)	串 string 右对齐，前面填充0
string.isdecimal()	isdecimal()方法检查字符串是否只包含十进制字符。这种方法只存在于unicode对象。

Python 列表(Lists)

序列是Python中最基本的数据结构。序列中的每个元素都分配一个数字 - 它的位置，或索引，第一个索引是0，第二个索引是1，依此类推。

Python有6个序列的内置类型，但最常见的是列表和元组。

序列都可以进行的操作包括索引，切片，加，乘，检查成员。

此外，Python已经内置确定序列的长度以及确定最大和最小的元素的方法。

列表是最常用的Python数据类型，它可以作为一个方括号内的逗号分隔值出现。

列表的数据项不需要具有相同的类型

创建一个列表，只要把逗号分隔的不同的数据项使用方括号括起来即可。如下所示：

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

与字符串的索引一样，列表索引从0开始。列表可以进行截取、组合等。

访问列表中的值

使用下标索引来访问列表中的值，同样你也可以使用方括号的形式截取字符，如下所示：

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];

print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

以上实例输出结果：

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

更新列表

你可以对列表的数据项进行修改或更新，你也可以使用append()方法来添加列表项，如下所示：

```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2 : "
print list[2];
list[2] = 2001;
print "New value available at index 2 : "
print list[2];
```

注意：我们会在接下来的章节讨论append()方法的使用

以上实例输出结果：

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

删除列表元素

可以使用 del 语句来删除列表的的元素，如下实例：

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];

print list1;
del list1[2];
print "After deleting value at index 2 : "
print list1;
```

以上实例输出结果：

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

注意：我们会在接下来的章节讨论remove()方法的使用

Python列表脚本操作符

列表对 + 和 的操作符与字符串相似。+ 号用于组合列表， 号用于重复列表。

如下所示：

Python 表达式	结果	描述
len([1, 2, 3])	3	长度
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	组合
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	重复
3 in [1, 2, 3]	True	元素是否存在于列表中
for x in [1, 2, 3]: print x,	1 2 3	迭代

Python列表截取

Python的列表截取与字符串操作类型，如下所示：

```
L = ['spam', 'Spam', 'SPAM!']
```

操作：

Python 表达式	结果	描述
L[2]	'SPAM!'	读取列表中第三个元素
L[-2]	'Spam'	读取列表中倒数第二个元素
L[1:]	['Spam', 'SPAM!']	从第二个元素开始截取列表

Python列表函数&方法

Python包含以下函数:

函数	描述
cmp(list1, list2)	比较两个列表的元素
len(list)	列表元素个数
max(list)	返回列表元素最大值
min(list)	返回列表元素最小值
list(seq)	将元组转换为列表

Python包含以下方法:

方法	描述
list.append(obj)	在列表末尾添加新的对象
list.count(obj)	统计某个元素在列表中出现的次数
list.extend(seq)	在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
list.index(obj)	从列表中找出某个值第一个匹配项的索引位置
list.insert(index, obj)	将对象插入列表
list.pop(obj=list[-1])	移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
list.remove(obj)	移除列表中某个值的第一个匹配项
list.reverse()	反向列表中元素
list.sort([func])	对原列表进行排序

Python 元组

Python的元组与列表类似，不同之处在于元组的元素不能修改。

元组使用小括号，列表使用方括号。

元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

如下实例：

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

创建空元组

```
tup1 = ();
```

元组中只包含一个元素时，需要在元素后面添加逗号

```
tup1 = (50,);
```

元组与字符串类似，下标索引从0开始，可以进行截取，组合等。

访问元组

元组可以使用下标索引来访问元组中的值，如下实例:

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0]
print "tup2[1:5]: ", tup2[1:5]
```

以上实例输出结果：

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

修改元组

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，如下实例:

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# 以下修改元组元素操作是非法的。
# tup1[0] = 100;

# 创建一个新的元组
tup3 = tup1 + tup2;
print tup3;
```

以上实例输出结果：

```
(12, 34.56, 'abc', 'xyz')
```

删除元组

元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组，如下实例:

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print tup;
del tup;
print "After deleting tup : "
print tup;
```

以上实例元组被删除后，输出变量会有异常信息，输出如下所示：

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

元组运算符

与字符串一样，元组之间可以使用 + 号和 * 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。

Python 表达式	结果	描述
len((1, 2, 3))	3	计算元素个数
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	连接
['Hi!'] * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	复制
3 in (1, 2, 3)	True	元素是否存在
for x in (1, 2, 3): print x,	1 2 3	迭代

元组索引，截取

因为元组也是一个序列，所以我们可以访问元组中的指定位置的元素，也可以截取索引中的一段元素，如下所示：

元组：

```
L = ('spam', 'Spam', 'SPAM!')
```

Python 表达式	结果	描述
L[2]	'SPAM!'	读取第三个元素
L[-2]	'Spam'	反向读取；读取倒数第二个元素
L[1:]	['Spam', 'SPAM!']	截取元素

无关闭分隔符

任意无符号的对象，以逗号隔开，默认为元组，如下实例：

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';
x, y = 1, 2;
print "Value of x , y : ", x,y;
```

以上实例允许结果：

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

元组内置函数

Python元组包含了以下内置函数

方法	描述
cmp(tuple1, tuple2)	比较两个元组元素。
len(tuple)	计算元组元素个数。
max(tuple)	返回元组中元素最大值。
min(tuple)	返回元组中元素最小值。
tuple(seq)	将列表转换为元组。

Python 字典(Dictionary)

字典是另一种可变容器模型，且可存储任意类型对象，如其他容器模型。

字典由键和对应值成对组成。字典也被称作关联数组或哈希表。基本语法如下：

```
dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

也可如此创建字典：

```
dict1 = { 'abc': 456 };
dict2 = { 'abc': 123, 98.6: 37 };
```

每个键与值用冒号隔开（:），每对用逗号，每对用逗号分割，整体放在花括号中（{}）。

键必须独一无二，但值则不必。

值可以取任何数据类型，但必须是不可变的，如字符串，数或元组。

访问字典里的值

把相应的键放入熟悉的方括弧，如下实例:

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Name']: ", dict['Name'];
print "dict['Age']: ", dict['Age'];
```

以上实例输出结果：

```
dict['Name']:  Zara
dict['Age']:  7
```

如果用字典里没有的键访问数据，会输出错误如下：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Alice']: ", dict['Alice'];
```

以上实例输出结果：

```
dict['Zara']:
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

修改字典

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对如下实例:

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age'];
print "dict['School']: ", dict['School'];
```

以上实例输出结果：

```
dict['Age']:  8
dict['School']:  DPS School
```

删除字典元素

能删单一的元素也能清空字典，清空只需一项操作。

显示删除一个字典用del命令，如下实例：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

del dict['Name']; # 删除键是'Name'的条目
dict.clear();     # 清空词典所有条目
del dict ;        # 删除词典

print "dict['Age']: ", dict['Age'];
print "dict['School']: ", dict['School'];
```

但这会引发一个异常，因为用del后字典不再存在：

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

注：del()方法后面也会讨论。

删除字典元素

字典键的特性

字典值可以没有限制地取任何python对象，既可以是标准的对象，也可以是用户定义的，但键不行。

两个重要的点需要记住：

1）不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住，如下实例：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};

print "dict['Name']: ", dict['Name'];
```

以上实例输出结果：

```
dict['Name']:  Manni
```

2）键必须不可变，所以可以用数，字符串或元组充当，所以用列表就不行，如下实例：

```
#!/usr/bin/python

dict = {['Name']: 'Zara', 'Age': 7};

print "dict['Name']: ", dict['Name'];
```

以上实例输出结果：

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    dict = {['Name']: 'Zara', 'Age': 7};
TypeError: list objects are unhashable
```

字典内置函数&方法

Python字典包含了以下内置函数：

函数	描述
cmp(dict1, dict2)	比较两个字典元素。
len(dict)	计算字典元素个数，即键的总数。
str(dict)	输出字典可打印的字符串表示。
type(variable)	返回输入的变量类型，如果变量是字典就返回字典类型。

Python字典包含了以下内置函数：

序号	函数及描述
radiansdict.clear()	删除字典内所有元素
radiansdict.copy()	返回一个字典的浅复制
radiansdict.fromkeys()	创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值
radiansdict.get(key, default=None)	返回指定键的值，如果值不在字典中返回default值
radiansdict.has_key(key)	如果键在字典dict里返回true，否则返回false
radiansdict.items()	以列表返回可遍历的(键, 值) 元组数组
radiansdict.keys()	以列表返回一个字典所有的键
radiansdict.setdefault(key, default=None)	和get()类似, 但如果键不已经存在于字典中，将会添加键并将值设为default
radiansdict.update(dict2)	把字典dict2的键/值对更新到dict里
radiansdict.values()	以列表返回字典中的所有值

Python 日期和时间

Python程序能用很多方式处理日期和时间。转换日期格式是一个常见的例行琐事。Python有一个time and calendar模组可以帮忙。

什么是Tick？

时间间隔是以秒为单位的浮点小数。

每个时间戳都以自从1970年1月1日午夜（历元）经过了多长时间来表示。

Python附带的受欢迎的time模块下有很多函数可以转换常见日期格式。如函数time.time()用ticks计时单位返回从12:00am, January 1, 1970(epoch) 开始的记录的当前操作系统时间, 如下实例:

```
#!/usr/bin/python
import time;  # This is required to include time module.

ticks = time.time()
print "Number of ticks since 12:00am, January 1, 1970:", ticks
```

以上实例输出结果：

```
Number of ticks since 12:00am, January 1, 1970: 7186862.73399
```

Tick单位最适于做日期运算。但是1970年之前的日期就无法以此表示了。太遥远的日期也不行，UNIX和Windows只支持到2038年某日。

什么是时间元组？

很多Python函数用一个元组装起来的9组数字处理时间:

字段	值
4位数年	2008
月	1 到 12
日	1到31
小时	0到23
分钟	
秒	0到61 (60或61 是闰秒)
一周的第几日	0到6 (0是周一)
一年的第几日	1到366 (儒略历)
夏令时	-1, 0, 1, -1是决定是否为夏令时的旗帜

上述也就是struct_time元组。这种结构具有如下属性：

属性	值
tm_year	2008
tm_mon	1 到 12
tm_mday	1 到 31
tm_hour	0 到 23
tm_min	0 到 59
tm_sec	0 到 61 (60或61 是闰秒)
tm_wday	0到6 (0是周一)
tm_yday	1 到 366(儒略历)
tm_isdst	-1, 0, 1, -1是决定是否为夏令时的旗帜

获取当前时间

从返回浮点数的时间辍方式向时间元组转换，只要将浮点数传递给如localtime之类的函数。

```
#!/usr/bin/python
import time;

localtime = time.localtime(time.time())
print "Local current time :", localtime
```

以上实例输出结果：

```
Local current time : time.struct_time(tm_year=2013, tm_mon=7,
tm_mday=17, tm_hour=21, tm_min=26, tm_sec=3, tm_wday=2, tm_yday=198
```

获取格式化的时间

你可以根据需求选取各种格式，但是最简单的获取可读的时间模式的函数是asctime():

```
#!/usr/bin/python
import time;

localtime = time.asctime( time.localtime(time.time()) )
print "Local current time :", localtime
```

以上实例输出结果：

```
Local current time : Tue Jan 13 10:17:09 2009
```

获取某月日历

Calendar模块有很广泛的方法用来处理年历和月历，例如打印某月的月历：

```
#!/usr/bin/python
import calendar

cal = calendar.month(2008, 1)
print "Here is the calendar:"
print cal;
```

以上实例输出结果：

```
Here is the calendar:
    January 2008
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

Time模块

Time模块包含了以下内置函数，既有时间处理相的，也有转换时间格式的：

函数	描述
time.altzone	返回格林威治西部的夏令时地区的偏移秒数。如果该地区在格林威治东部会返回负值（如西欧，包括英国）。对夏令时启用地区才能使用。
time.asctime([tupletime])	接受时间元组并返回一个可读的形式为"Tue Dec 11 18:07:14 2008"（2008年12月11日 周二18时07分14秒）的24个字符的字符串。
time.clock( )	用以浮点数计算的秒数返回当前的CPU时间。用来衡量不同程序的耗时，比time.time()更有用。
time.ctime([secs])	作用相当于asctime(localtime(secs))，未给参数相当于asctime()
time.gmtime([secs])	接收时间辍（1970纪元后经过的浮点秒数）并返回格林威治天文时间下的时间元组t。注：t.tm_isdst始终为0
time.localtime([secs])	接收时间辍（1970纪元后经过的浮点秒数）并返回当地时间下的时间元组t（t.tm_isdst可取0或1，取决于当地当时是不是夏令时）。
time.mktime(tupletime)	接受时间元组并返回时间辍（1970纪元后经过的浮点秒数）。
time.sleep(secs)	推迟调用线程的运行，secs指秒数。
time.strftime(fmt[,tupletime])	接收以时间元组，并返回以可读字符串表示的当地时间，格式由fmt决定。
time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')	根据fmt的格式把一个时间字符串解析为时间元组。
time.time( )	返回当前时间的时间戳（1970纪元后经过的浮点秒数）。
time.tzset()	根据环境变量TZ重新初始化时间相关设置。

Time模块包含了以下2个非常重要的属性：

属性	描述
time.timezone	属性time.timezone是当地时区（未启动夏令时）距离格林威治的偏移秒数（>0，美洲;<=0大部分欧洲，亚洲，非洲）。
time.tzname	属性time.tzname包含一对根据情况的不同而不同的字符串，分别是带夏令时的本地时区名称，和不带的。

日历（Calendar）模块

此模块的函数都是日历相关的，例如打印某月的字符月历。

星期一是默认的每周第一天，星期天是默认的最后一天。更改设置需调用calendar.setfirstweekday()函数。模块包含了以下内置函数：

序号	函数及描述
calendar.calendar(year,w=2,l=1,c=6)	返回一个多行字符串格式的year年年历，3个月一行，间隔距离为c。每日宽度间隔为w字符。每行长度为21 *W+18+*2 C。l是每星期行数。
calendar.firstweekday( )	返回当前每周起始日期的设置。默认情况下，首次载入caendar模块时返回0，即星期一。
calendar.isleap(year)	是闰年返回True，否则为false。
calendar.leapdays(y1,y2)	返回在Y1，Y2两年之间的闰年总数。
calendar.month(year,month,w=2,l=1)	返回一个多行字符串格式的year年month月日历，两行标题，一周一行。每日宽度间隔为w字符。每行的长度为7* w+6。l是每星期的行数。
calendar.monthcalendar(year,month)	返回一个整数的单层嵌套列表。每个子列表装载代表一个星期的整数。Year年month月外的日期都设为0;范围内的日子都由该月第几日表示，从1开始。
calendar.monthrange(year,month)	返回两个整数。第一个是该月的星期几的日期码，第二个是该月的日期码。日从0（星期一）到6（星期日）;月从1到12。
calendar.prcal(year,w=2,l=1,c=6)	相当于 print

calendar.prcal(year,w=2,l=1,c=6)	calendar.calendar(year,w,l,c).
calendar.prmonth(year,month,w=2,l=1)	相当于 print calendar.calendar（year，w，l，c）。
calendar.setfirstweekday(weekday)	设置每周的起始日期码。0（星期一）到6（星期日）。
calendar.timegm(tupletime)	和time.gmtime相反：接受一个时间元组形式，返回该时刻的时间辍（1970纪元后经过的浮点秒数）。
calendar.weekday(year,month,day)	返回给定日期的日期码。0（星期一）到6（星期日）。月份为 1（一月） 到 12（12月）。

其他相关模块和函数

在Python种，其他处理日期和时间的模块还有：

- datetime模块
- pytz模块
- ateutil模块

Python函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。你已经知道Python提供了许多内建函数，比如print()。但你也可以自己创建函数，这被叫做用户自定义函数。

定义一个函数

你可以定义一个由自己想要功能的函数，以下是简单的规则：

- 函数代码块以def关键词开头，后接函数标识符名称和圆括号()。
- 任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于定义参数。
- 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
- 函数内容以冒号起始，并且缩进。
- Return[expression]结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 None。

语法

```
def functionname( parameters ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

默认情况下，参数值和参数名称是按函数声明中定义的的顺序匹配起来的。

实例

以下为一个简单的Python函数，它将一个字符串作为传入参数，再打印到标准显示设备上。

```
def printme( str ):
   "打印传入的字符串到标准显示设备上"
   print str
   return
```

函数调用

定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。

这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从Python提示符执行。

如下实例调用了printme（）函数：

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;

# Now you can call printme function
printme("我要调用用户自定义函数!");
printme("再次调用同一函数");
```

以上实例输出结果：

```
我要调用用户自定义函数!
再次调用同一函数
```

按值传递参数和按引用传递参数

所有参数（自变量）在Python里都是按引用传递。如果你在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。例如：

```
#!/usr/bin/python

# 可写函数说明
def changeme( mylist ):
   "修改传入的列表"
   mylist.append([1,2,3,4]);
   print "函数内取值: ", mylist
   return

# 调用changeme函数
mylist = [10,20,30];
changeme( mylist );
print "函数外取值: ", mylist
```

传入函数的和在末尾添加新内容的对象用的是同一个引用。故输出结果如下：

```
函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
函数外取值:  [10, 20, 30, [1, 2, 3, 4]]
```

参数

以下是调用函数时可使用的正式参数类型：

- 必备参数
- 命名参数
- 缺省参数
- 不定长参数

必备参数

必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。

调用printme()函数，你必须传入一个参数，不然会出现语法错误：

```
#!/usr/bin/python

#可写函数说明
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;

#调用printme函数
printme();
```

以上实例输出结果：

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

命名参数

命名参数和函数调用关系紧密，调用方用参数的命名确定传入的参数值。你可以跳过不传的参数或者乱序传参，因为Python解释器能够用参数名匹配参数值。用命名参数调用printme()函数：

```
#!/usr/bin/python

#可写函数说明
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;

#调用printme函数
printme( str = "My string");
```

以上实例输出结果：

```
My string
```

下例能将命名参数顺序不重要展示得更清楚：

```
#!/usr/bin/python

#可写函数说明
def printinfo( name, age ):
   "打印任何传入的字符串"
   print "Name: ", name;
   print "Age ", age;
   return;

#调用printinfo函数
printinfo( age=50, name="miki" );
```

以上实例输出结果：

```
Name:  miki
Age  50
```

缺省参数

调用函数时，缺省参数的值如果没有传入，则被认为是默认值。下例会打印默认的age，如果age没有被传入：

```
#!/usr/bin/python

#可写函数说明
def printinfo( name, age = 35 ):
   "打印任何传入的字符串"
   print "Name: ", name;
   print "Age ", age;
   return;

#调用printinfo函数
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

以上实例输出结果：

```
Name:  miki
Age  50
Name:  miki
Age  35
```

不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述2种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号（*）的变量名会存放所有未命名的变量参数。选择不多传参数也可。如下实例：

```
#!/usr/bin/python

# 可写函数说明
def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print "输出: "
   print arg1
   for var in vartuple:
      print var
   return;

# 调用printinfo 函数
printinfo( 10 );
printinfo( 70, 60, 50 );
```

以上实例输出结果：

```
输出:
10
输出:
70
60
50
```

匿名函数

用lambda关键词能创建小型匿名函数。这种函数得名于省略了用def声明函数的标准步骤。

- Lambda函数能接收任何数量的参数但只能返回一个表达式的值，同时只能不能包含命令或多个表达式。
- 匿名函数不能直接调用print，因为lambda需要一个表达式。
- lambda函数拥有自己的名字空间，且不能访问自有参数列表之外或全局名字空间里的参数。
- 虽然lambda函数看起来只能写一行，却不等同于C或C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

语法

lambda函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

如下实例：

```
#!/usr/bin/python

#可写函数说明
sum = lambda arg1, arg2: arg1 + arg2;

#调用sum函数
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

以上实例输出结果：

```
Value of total :  30
Value of total :  40
```

return语句

return语句[表达式]退出函数，选择性地向调用方返回一个表达式。不带参数值的return语句返回None。之前的例子都没有示范如何返回数值，下例便告诉你怎么做：

```
#!/usr/bin/python

# 可写函数说明
def sum( arg1, arg2 ):
   # 返回2个参数的和."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# 调用sum函数
total = sum( 10, 20 );
print "Outside the function : ", total
```

以上实例输出结果：

```
Inside the function :  30
Outside the function :  30
```

变量作用域

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。两种最基本的变量作用域如下：

- 全局变量
- 局部变量

变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。如下实例：

```
#!/usr/bin/python

total = 0; # This is global variable.
# 可写函数说明
def sum( arg1, arg2 ):
   #返回2个参数的和."
   total = arg1 + arg2; # total在这里是局部变量.
   print "Inside the function local total : ", total
   return total;

#调用sum函数
sum( 10, 20 );
print "Outside the function global total : ", total
```

以上实例输出结果：

```
Inside the function local total :  30
Outside the function global total :  0
```

Python 模块

模块让你能够有逻辑地组织你的Python代码段。

把相关的代码分配到一个 模块里能让你的代码更好用，更易懂。

模块也是Python对象，具有随机的名字属性用来绑定或引用。

简单地说，模块就是一个保存了Python代码的文件。模块能定义函数，类和变量。模块里也能包含可执行的代码。

例子

一个叫做aname的模块里的Python代码一般都能在一个叫aname.py的文件中找到。下例是个简单的模块support.py。

```
def print_func( par ):
   print "Hello : ", par
   return
```

import 语句

想使用Python源文件，只需在另一个源文件里执行import语句，语法如下：

```
import module1[, module2[,... moduleN]
```

当解释器遇到import语句，如果模块在当前的搜索路径就会被导入。

搜索路径是一个解释器会先进行搜索的所有目录的列表。如想要导入模块hello.py，需要把命令放在脚本的顶端：

```
#!/usr/bin/python

# 导入模块
import support

# 现在可以调用模块里包含的函数了
support.print_func("Zara")
```

以上实例输出结果：

```
Hello : Zara
```

一个模块只会被导入一次，不管你执行了多少次import。这样可以防止导入模块被一遍又一遍地执行。

From…import 语句

Python的from语句让你从模块中导入一个指定的部分到当前命名空间中。语法如下：

```
from modname import name1[, name2[, ... nameN]]
```

例如，要导入模块fib的fibonacci函数，使用如下语句：

```
from fib import fibonacci
```

这个声明不会把整个fib模块导入到当前的命名空间中，它只会将fib里的fibonacci单个引入到执行这个声明的模块的全局符号表。

From…import* 语句

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

```
from modname import *
```

这提供了一个简单的方法来导入一个模块中的所有项目。然而这种声明不该被过多地使用。

定位模块

当你导入一个模块，Python解析器对模块位置的搜索顺序是：

- 当前目录
- 如果不在当前目录，Python则搜索在shell变量PYTHONPATH下的每个目录
- 如果都找不到，Python会察看默认路径。UNIX下，默认路径一般为/usr/local/lib/python/

模块搜索路径存存储在system模块的sys.path变量中。变量里包含当前目录，PYTHONPATH和由安装过程决定的默认目录。

PYTHONPATH变量

作为环境变量，PYTHONPATH由装在一个列表里的许多目录组成。PYTHONPATH的语法和shell变量PATH的一样。

在Windows系统，典型的PYTHONPATH如下：

```
set PYTHONPATH=c:\python20\lib;
```

在UNIX系统，典型的PYTHONPATH如下：

```
set PYTHONPATH=/usr/local/lib/python
```

命名空间和作用域

变量是拥有匹配对象的名字（标识符）。命名空间是一个包含了变量名称们（键）和它们各自相应的对象们（值）的字典。

一个Python表达式可以访问局部命名空间和全局命名空间里的变量。如果一个局部变量和一个全局变量重名，则局部变量会覆盖全局变量。

每个函数都有自己的命名空间。类的方法的作用域规则和通常函数的一样。

Python会智能地猜测一个变量是局部的还是全局的，它假设任何在函数内赋值的变量都是局部的。

因此，如果要给全局变量在一个函数里赋值，必须使用global语句。

global VarName的表达式会告诉Python， VarName是一个全局变量，这样Python就不会在局部命名空间里寻找这个变量了。

例如，我们在全局命名空间里定义一个变量money。我们再在函数内给变量money赋值，然后Python会假定money是一个局部变量。然而，我们并没有在访问前声明一个局部变量money，结果就是会出现一个UnboundLocalError的错误。取消global语句的注释就能解决这个问题。

```
#!/usr/bin/python

Money = 2000
def AddMoney():
   # 想改正代码就取消以下注释:
   # global Money
   Money = Money + 1

print Money
AddMoney()
print Money
```

dir()函数

dir()函数一个排好序的字符串列表，内容是一个模块里定义过的名字。

返回的列表容纳了在一个模块里定义的所有模块，变量和函数。如下一个简单的实例：

```
#!/usr/bin/python

# 导入内置math模块
import math

content = dir(math)

print content;
```

以上实例输出结果：

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

在这里，特殊字符串变量**name**指向模块的名字，**file**指向该模块的导入文件名。

globals()和locals()函数

根据调用地方的不同，globals()和locals()函数可被用来返回全局和局部命名空间里的名字。

如果在函数内部调用locals()，返回的是所有能在该函数里访问的命名。

如果在函数内部调用globals()，返回的是所有在该函数里能访问的全局名字。

两个函数的返回类型都是字典。所以名字们能用keys()函数摘取。

reload()函数

当一个模块被导入到一个脚本，模块顶层部分的代码只会被执行一次。

因此，如果你想重新执行模块里顶层部分的代码，可以用reload()函数。该函数会重新导入之前导入过的模块。语法如下：

```
reload(module_name)
```

在这里，module_name要直接放模块的名字，而不是一个字符串形式。比如想重载hello模块，如下：

```
reload(hello)
```

Python中的包

包是一个分层次的文件目录结构，它定义了一个由模块及子包，和子包下的子包等组成的Python的应用环境。

考虑一个在Phone目录下的pots.py文件。这个文件有如下源代码：

```
#!/usr/bin/python

def Pots():
   print "I'm Pots Phone"
```

同样地，我们有另外两个保存了不同函数的文件：

- Phone/Isdn.py 含有函数Isdn()
- Phone/G3.py 含有函数G3()

现在，在Phone目录下创建file **init**.py：

- Phone/**init**.py

当你导入Phone时，为了能够使用所有函数，你需要在**init**.py里使用显式的导入语句，如下：

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

当你把这些代码添加到**init**.py之后，导入Phone包的时候这些类就全都是可用的了。

```
#!/usr/bin/python

# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

以上实例输出结果：

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

如上，为了举例，我们只在每个文件里放置了一个函数，但其实你可以放置许多函数。你也可以在这些文件里定义Python的类，然后为这些类建一个包。

Python 文件I/O

本章只讲述所有基本的的I/O函数，更多函数请参考Python标准文档。

打印到屏幕

最简单的输出方法是用print语句，你可以给它传递零个或多个用逗号隔开的表达式。此函数把你传递的表达式转换成一个字符串表达式，并将结果写到标准输出如下：

```
#!/usr/bin/python

print "Python is really a great language,", "isn't it?";
```

你的标准屏幕上会产生以下结果：

```
Python is really a great language, isn't it?
```

读取键盘输入

Python提供了两个内置函数从标准输入读入一行文本，默认的标准输入是键盘。如下：

- raw_input
- input

raw_input函数

raw_input([prompt]) 函数从标准输入读取一个行，并返回一个字符串（去掉结尾的换行符）：

```
#!/usr/bin/python

str = raw_input("Enter your input: ");
print "Received input is : ", str
```

这将提示你输入任意字符串，然后在屏幕上显示相同的字符串。当我输入"Hello Python！"，它的输出如下：

```
Enter your input: Hello Python
Received input is :  Hello Python
```

input函数

input([prompt]) 函数和raw_input([prompt]) 函数基本可以互换，但是input会假设你的输入是一个有效的Python表达式，并返回运算结果。

```
#!/usr/bin/python

str = input("Enter your input: ");
print "Received input is : ", str
```

这会产生如下的对应着输入的结果：

```
Enter your input: [x*5 for x in range(2,10,2)]
Recieved input is :  [10, 20, 30, 40]
```

打开和关闭文件

到现在为止，您已经可以向标准输入和输进行读写。现在，来看看怎么读写实际的数据文件。

Python提供了必要的函数和方法进行默认情况下的文件基本操作。你可以用file对象做大部分的文件操作。

open函数

你必须先用Python内置的open()函数打开一个文件，创建一个file对象，相关的辅助方法才可以调用它进行读写。

语法：

```
file object = open(file_name [, access_mode][, buffering])
```

各个参数的细节如下：

- file_name：file_name变量是一个包含了你要访问的文件名称的字符串值。
- access_mode：access_mode决定了打开文件的模式：只读，写入，追加等。所有可取值见如下的完全列表。这个参数是非强制的，默认文件访问模式为只读(r)。
- buffering:如果buffering的值被设为0，就不会有寄存。如果buffering的值取1，访问文件时会寄存行。如果将buffering的值设为大于1的整数，表明了这就是

的寄存区的缓冲大小。如果取负值，寄存区的缓冲大小则为系统默认。

不同模式打开文件的完全列表：

模式	描述
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

File对象的属性

一个文件被打开后，你有一个file对象，你可以得到有关该文件的各种信息。

以下是和file对象相关的所有属性的列表：

属性	描述
file.closed	返回true如果文件已被关闭，否则返回false。
file.mode	返回被打开文件的访问模式。
file.name	返回文件的名称。
file.softspace	如果用print输出后，必须跟一个空格符，则返回false。否则返回true。

如下实例：

```
#!/usr/bin/python

# 打开一个文件
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

以上实例输出结果：

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

Close()方法

File对象的close（）方法刷新缓冲区里任何还没写入的信息，并关闭该文件，这之后便不能再进行写入。

当一个文件对象的引用被重新指定给另一个文件时，Python会关闭之前的文件。用close（）方法关闭文件是一个很好的习惯。

语法：

```
fileObject.close();
```

例子：

```
#!/usr/bin/python

# 打开一个文件
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# 关闭打开的文件
fo.close()
```

以上实例输出结果：

```
Name of the file:  foo.txt
```

读写文件：

file对象提供了一系列方法，能让我们的文件访问更轻松。来看看如何使用read()和write()方法来读取和写入文件。

Write()方法

Write()方法可将任何字符串写入一个打开的文件。需要重点注意的是，Python字符串可以是二进制数据，而不是仅仅是文字。

Write()方法不在字符串的结尾不添加换行符('\n')：

语法：

```
fileObject.write(string);
```

在这里，被传递的参数是要写入到已打开文件的内容。

例子：

```
#!/usr/bin/python

# 打开一个文件
fo = open("/tmp/foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n");

# 关闭打开的文件
fo.close()
```

上述方法会创建foo.txt文件，并将收到的内容写入该文件，并最终关闭文件。如果你打开这个文件，将看到以下内容:

```
Python is a great language.
Yeah its great!!
```

read()方法

read（）方法从一个打开的文件中读取一个字符串。需要重点注意的是，Python字符串可以是二进制数据，而不是仅仅是文字。

语法：

```
fileObject.read([count]);
```

在这里，被传递的参数是要从已打开文件中读取的字节计数。该方法从文件的开头开始读入，如果没有传入count，它会尝试尽可能多地读取更多的内容，很可能是直到文件的末尾。

例子：

就用我们上面创建的文件foo.txt。

```
#!/usr/bin/python

# 打开一个文件
fo = open("/tmp/foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# 关闭打开的文件
fo.close()
```

以上实例输出结果：

```
Read String is :  Python is
```

文件位置：

Tell()方法告诉你文件内的当前位置；换句话说，下一次的读写会发生在文件开头这么多字节之后：

seek（offset [,from]）方法改变当前文件的位置。Offset变量表示要移动的字节数。From变量指定开始移动字节的参考位置。

如果from被设为0，这意味着将文件的开头作为移动字节的参考位置。如果设为1，则使用当前的位置作为参考位置。如果它被设为2，那么该文件的末尾将作为参考位置。

例子：

就用我们上面创建的文件foo.txt。

```
#!/usr/bin/python

# 打开一个文件
fo = open("/tmp/foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str

# 查找当前位置
position = fo.tell();
print "Current file position : ", position

# 把指针再次重新定位到文件开头
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
# 关闭打开的文件
fo.close()
```

以上实例输出结果：

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

重命名和删除文件

Python的os模块提供了帮你执行文件处理操作的方法，比如重命名和删除文件。

要使用这个模块，你必须先导入它，然后可以调用相关的各种功能。

rename()方法：

rename()方法需要两个参数，当前的文件名和新文件名。

语法：

```
os.rename(current_file_name, new_file_name)
```

例子：

下例将重命名一个已经存在的文件test1.txt。

```
#!/usr/bin/python
import os

# 重命名文件test1.txt到test2.txt。
os.rename( "test1.txt", "test2.txt" )
```

remove()方法

你可以用remove()方法删除文件，需要提供要删除的文件名作为参数。

语法：

```
os.remove(file_name)
```

例子：

下例将删除一个已经存在的文件test2.txt。

```
#!/usr/bin/python
import os

# 删除一个已经存在的文件test2.txt
os.remove("text2.txt")
```

Python里的目录：

所有文件都包含在各个不同的目录下，不过Python也能轻松处理。os模块有许多方法能帮你创建，删除和更改目录。

mkdir()方法

可以使用os模块的mkdir()方法在当前目录下创建新的目录们。你需要提供一个包含了要创建的目录名称的参数。

语法：

```
os.mkdir("newdir")
```

例子：

下例将在当前目录下创建一个新目录test。

```
#!/usr/bin/python
import os

# 创建目录test
os.mkdir("test")
```

chdir()方法

可以用chdir()方法来改变当前的目录。chdir()方法需要的一个参数是你想设成当前目录的目录名称。

语法：

```
os.chdir("newdir")
```

例子：

下例将进入"/home/newdir"目录。

```
#!/usr/bin/python
import os

# 将当前目录改为"/home/newdir"
os.chdir("/home/newdir")
```

getcwd()方法：

getcwd()方法显示当前的工作目录。

语法：

```
os.getcwd()
```

例子：

下例给出当前目录：

```
#!/usr/bin/python
import os

# 给出当前的目录
os.getcwd()
```

rmdir()方法

rmdir()方法删除目录，目录名称以参数传递。

在删除这个目录之前，它的所有内容应该先被清除。

语法：

```
os.rmdir('dirname')
```

例子：

以下是删除" /tmp/test"目录的例子。目录的完全合规的名称必须被给出，否则会在当前目录下搜索该目录。

```
#!/usr/bin/python
import os

# 删除”/tmp/test”目录
os.rmdir( "/tmp/test"  )
```

文件、目录相关的方法

三个重要的方法来源能对Windows和Unix操作系统上的文件及目录进行一个广泛且实用的处理及操控，如下：

- File 对象方法: file对象提供了操作文件的一系列方法。
- OS 对象方法: 提供了处理文件及目录的一系列方法。

Python 异常处理

python提供了两个非常重要的功能来处理python程序在运行中出现的异常和错误。你可以使用该功能来调试python程序。

- 异常处理: 本站Python教程会具体介绍。
- 断言(Assertions):本站Python教程会具体介绍。

python标准异常

异常名称	描述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入^C)
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
SystemExit	Python 解释器请求退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零 (所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败

KeyboardInterrupt	用户中断执行(通常是输入^C)
LookupError	无效数据查询的基类
IndexError	序列中没有没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于Python 解释器不是致命的)
NameError	未声明/初始化对象 (没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

什么是异常？

异常即是一个事件，该事件会在程序执行过程中发生，影响了程序的正常执行。

一般情况下，在Python无法正常处理程序时就会发生一个异常。

异常是Python对象，表示一个错误。

当Python脚本发生异常时我们需要捕获处理它，否则程序会终止执行。

异常处理

捕捉异常可以使用try/except语句。

try/except语句用来检测try语句块中的错误，从而让except语句捕获异常信息并处理。

如果你不想在异常发生时结束你的程序，只需在try里捕获它。

语法：

以下为简单的*try....except...else*的语法：

```
try:
<语句>        #运行别的代码
except <名字>：
<语句>        #如果在try部份引发了'name'异常
except <名字>，<数据>:
<语句>        #如果引发了'name'异常，获得附加的数据
else:
<语句>        #如果没有异常发生
```

try的工作原理是，当开始一个try语句后，python就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，try子句先执行，接下来会发生什么依赖于执行时是否出现异常。

- 如果当try后的语句执行时发生异常，python就跳回到try并执行第一个匹配该异常的except子句，异常处理完毕，控制流就通过整个try语句（除非在处理异常时又引发新的异常）。
- 如果在try后的语句里发生了异常，却没有匹配的except子句，异常将被递交到上层的try，或者到程序的最上层（这样将结束程序，并打印缺省的出错信息）。
- 如果在try子句执行时没有发生异常，python将执行else语句后的语句（如果有else的话），然后控制流通过整个try语句。

实例

下面是简单的例子，它打开一个文件，在该文件中的内容写入内容，且并未发生异常：

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

以上程序输出结果：

```
 Written content in the file successfully
```

实例

下面是简单的例子，它打开一个文件，在该文件中的内容写入内容，但文件没有写入权限，发生了异常：

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

以上程序输出结果：

```
Error: can't find file or read data
```

使用except而不带任何异常类型

你可以不带任何异常类型使用except，如下实例：

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

以上方式try-except语句捕获所有发生的异常。但这不是一个很好的方式，我们不能通过该程序识别出具体的异常信息。因为它捕获所有的异常。

使用except而带多种异常类型

你也可以使用相同的except语句来处理多个异常信息，如下所示：

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

try-finally 语句

try-finally 语句无论是否发生异常都将执行最后的代码。

```
try:
<语句>
finally:
<语句>    #退出try时总会执行
raise
```

注意：你可以使用except语句或者finally语句，但是两者不能同时使用。else语句也不能与finally语句同时使用

实例

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print "Error: can\'t find file or read data"
```

如果打开的文件没有可写权限，输出如下所示：

```
Error: can't find file or read data
```

同样的例子也可以写成如下方式：

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print "Going to close the file"
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

当在try块中抛出一个异常，立即执行finally块代码。

finally块中的所有语句执行后，异常被再次提出，并执行except块代码。

参数的内容不同于异常。

异常的参数

一个异常可以带上参数，可作为输出的异常信息参数。

你可以通过except语句来捕获异常的参数，如下所示：

```
try:
   You do your operations here;
   ......................
except ExceptionType, Argument:
   You can print value of Argument here...
```

变量接收的异常值通常包含在异常的语句中。在元组的表单中变量可以接收一个或者多个值。

元组通常包含错误字符串，错误数字，错误位置。

实例

以下为单个异常的实例：

```
#!/usr/bin/python

# Define a function here.
def temp_convert(var):
   try:
      return int(var)
   except ValueError, Argument:
      print "The argument does not contain numbers\n", Argument

# Call above function here.
temp_convert("xyz");
```

以上程序执行结果如下：

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

触发异常

我们可以使用raise语句自己触发异常

raise语法格式如下：

```
raise [Exception [, args [, traceback]]]
```

语句中Exception是异常的类型（例如，NameError）参数是一个异常参数值。该参数是可选的，如果不提供，异常的参数是"None"。

最后一个参数是可选的（在实践中很少使用），如果存在，是跟踪异常对象。

实例

一个异常可以是一个字符串，类或对象。 Python的内核提供的异常，大多数都是实例化的类，这是一个类的实例的参数。

定义一个异常非常简单，如下所示：

```
def functionName( level ):
   if level < 1:
      raise "Invalid level!", level
      # The code below to this would not be executed
      # if we raise the exception
```

注意：为了能够捕获异常，"except"语句必须有用相同的异常来抛出类对象或者字符串。

例如我们捕获以上异常，"except"语句如下所示：

```
try:
   Business Logic here...
except "Invalid level!":
   Exception handling here...
else:
   Rest of the code here...
```

用户自定义异常

通过创建一个新的异常类，程序可以命名它们自己的异常。异常应该是典型的继承自Exception类，通过直接或间接的方式。

以下为与RuntimeError相关的实例,实例中创建了一个类，基类为RuntimeError，用于在异常触发时输出更多的信息。

在try语句块中，用户自定义的异常后执行except块语句，变量 e 是用于创建Networkerror类的实例。

```
class Networkerror(RuntimeError):
   def __init__(self, arg):
      self.args = arg
```

在你定义以上类后，你可以触发该异常，如下所示：

```
try:
   raise Networkerror("Bad hostname")
except Networkerror,e:
   print e.args
```

Python 高级教程

Python面向对象

Python从设计之初就已经是一门面向对象的语言，正因为如此，在Python中创建一个类和对象是很容易的。本章节我们将详细介绍Python的面向对象编程。

如果你以前没有接触过面向对象的编程语言，那你可能需要先了解一些面向对象语言的一些基本特征，在头脑里头形成一个基本的面向对象的概念，这样有助于你更容易的学习Python的面向对象编程。

接下来我们先来简单的了解下面向对象的一些基本特征。

面向对象技术简介

- 类**(Class):** 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- 类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- 数据成员：类变量或者实例变量用于处理类及其实例对象的相关的数据。
- 方法重载：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重载。
- 实例变量：定义在方法中的变量，只作用于当前实例的类。
- 继承：即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个Dog类型的对象派生自Animal类，这是模拟"是一个（is-a）"关系（例图，Dog是一个Animal）。
- 实例化：创建一个类的实例，类的具体对象。
- 方法：类中定义的函数。
- 对象：通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

创建类

使用class语句来创建一个新类，class之后为类的名称并以冒号结尾，如下实例:

```
class ClassName:
   'Optional class documentation string'#类文档字符串
   class_suite  #类体
```

类的帮助信息可以通过ClassName.**doc**查看。

class_suite 由类成员，方法，数据属性组成。

实例

以下是一个简单的Python类实例:

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- empCount变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用Employee.empCount访问。
- 第一种方法**init**()方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法

创建实例对象

要创建一个类的实例，你可以使用类的名称，并通过**init**方法接受参数。

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

访问属性

您可以使用点(.)来访问对象的属性。使用如下类的名称访问类变量:

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

完整实例：

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

执行以上代码输出结果如下：

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

你可以添加，删除，修改类的属性，如下所示：

```
emp1.age = 7  # 添加一个 'age' 属性
emp1.age = 8  # 修改 'age' 属性
del emp1.age  # 删除 'age' 属性
```

你也可以使用以下函数的方式来访问属性：

- getattr(obj, name[, default]) : 访问对象的属性。
- hasattr(obj,name) : 检查是否存在一个属性。
- setattr(obj,name,value) : 设置一个属性。如果属性不存在，会创建一个新属性。
- delattr(obj, name) : 删除属性。

```
hasattr(emp1, 'age')    # 如果存在 'age' 属性返回 True。
getattr(emp1, 'age')    # 返回 'age' 属性的值
setattr(emp1, 'age', 8) # 添加属性 'age' 值为 8
delattr(empl, 'age')    # 删除属性 'age'
```

Python内置类属性

- **dict** : 类的属性（包含一个字典，由类的数据属性组成）
- **doc** :类的文档字符串
- **name**: 类名
- **module**: 类定义所在的模块（类的全名是'**main**.className'，如果类位于一个导入模块mymod中，那么className.**module** 等于 mymod）
- **bases** : 类的所有父类构成元素（包含了以个由所有父类组成的元组）

Python内置类属性调用实例如下：

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

执行以上代码输出结果如下：

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

python对象销毁(垃圾回收)

同Java语言一样，Python使用了引用计数这一简单技术来追踪内存中的对象。

在Python内部记录着所有使用中的对象各有多少引用。

一个内部跟踪变量，称为一个引用计数器。

当对象被创建时， 就创建了一个引用计数， 当这个对象不再需要时， 也就是说，这个对象的引用计数变为0 时， 它被垃圾回收。但是回收不是"立即"的， 由解释器在适当的时机，将垃圾对象占用的内存空间回收。

```
a = 40      # 创建对象  <40>
b = a       # 增加引用， <40> 的计数
c = [b]     # 增加引用.  <40> 的计数

del a       # 减少引用 <40> 的计数
b = 100     # 减少引用 <40> 的计数
c[0] = -1   # 减少引用 <40> 的计数
```

垃圾回收机制不仅针对引用计数为0的对象，同样也可以处理循环引用的情况。循环引用指的是，两个对象相互引用，但是没有其他变量引用他们。这种情况下，仅使用引用计数是不够的。Python 的垃圾收集器实际上是一个引用计数器和一个循环垃圾收集器。作为引用计数的补充， 垃圾收集器也会留心被分配的总量很大（及未通过引用计数销毁的那些）的对象。 在这种情况下， 解释器会暂停下来， 试图清理所有未引用的循环。

实例

析构函数 **del** ， **del**在对象消逝的时候被调用，当对象不再被使用时，**del**方法运行：

```
#!/usr/bin/python

class Point:
   def __init( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # 打印对象的id
del pt1
del pt2
del pt3
<pre>
<p>以上实例运行结果如下：</p>
<pre>
3083401324 3083401324 3083401324
Point destroyed
```

注意：通常你需要在单独的文件中定义一个类，

类的继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。继承完全可以理解成类之间的类型和子类型关系。

需要注意的地方：继承语法 class 派生类名（基类名）：//... 基类名写作括号里，基本类是在类定义的时候，在元组之中指明的。

在python中继承中的一些特点：

- 1：在继承中基类的构造（**init**()方法）不会被自动调用，它需要在其派生类的构造中亲自专门调用。
- 2：在调用基类的方法时，需要加上基类的类名前缀，且需要带上self参数变量。区别于在类中调用普通函数时并不需要带上self参数
- 3：Python总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。（先在本类中查找调用的方法，找不到才去基类中找）。

如果在继承元组中列了一个以上的类，那么它就被称作"多重继承" 。

语法：

派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

实例：

```
#!/usr/bin/python

class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # 实例化子类
c.childMethod()      # 调用子类的方法
c.parentMethod()     # 调用父类方法
c.setAttr(200)       # 再次调用父类的方法
c.getAttr()          # 再次调用父类的方法
```

以上代码执行结果如下：

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

你可以继承多个类

```
class A:        # define your class A
.....

class B:         # define your calss B
.....

class C(A, B):   # subclass of A and B
.....
```

你可以使用issubclass()或者isinstance()方法来检测。

- issubclass() - 布尔函数判断一个类是另一个类的子类或者子孙类，语法：issubclass(sub,sup)
- isinstance(obj, Class) 布尔函数如果obj是Class类的实例对象或者是一个Class子类的实例对象则返回true。

重载方法

如果你的父类方法的功能不能满足你的需求，你可以在子类重载你父类的方法：

实例：

```
#!/usr/bin/python

class Parent:        # 定义父类
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # 定义子类
   def myMethod(self):
      print 'Calling child method'

c = Child()          # 子类实例
c.myMethod()         # 子类调用重载方法
```

执行以上代码输出结果如下：

```
Calling child method
```

基础重载方法

下表列出了一些通用的功能，你可以在自己的类重写：

方法	描述 & 简单的调用
__init__ (self [,args...])	构造函数 简单的调用方法: *obj = className(args)*
__del__(self)	析构方法, 删除一个对象 简单的调用方法 : *del obj*
__repr__(self)	转化为供解释器读取的形式 简单的调用方法 : *repr(obj)*
__str__(self)	用于将值转化为适于人阅读的形式 简单的调用方法 : *str(obj)*
__cmp__ (self, x)	对象比较 简单的调用方法 : *cmp(obj, x)*

运算符重载

Python同样支持运算符重载，实例如下：

```
#!/usr/bin/python

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

以上代码执行结果如下所示:

```
Vector(7,8)
```

隐藏数据

在python中实现数据隐藏很简单，不需要在前面加什么关键字，只要把类变量名或成员函数前面加两个下划线即可实现数据隐藏的功能，这样，对于类的实例来说，其变量名和成员函数是不能使用的，对于其类的继承类来说，也是隐藏的，这样，

其继承类可以定义其一模一样的变量名或成员函数名，而不会引起命名冲突。 实例：

```
#!/usr/bin/python

class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

Python 通过改变名称来包含类名:

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCour
```

Python不允许实例化的类访问隐藏数据，但你可以使用 object._className__attrName访问属性，将如下代码替换以上代码的最后一行代码：

```
.........................
print counter._JustCounter__secretCount
```

执行以上代码，执行结果如下：

```
1
2
2
```

Python正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。Python 自1.5版本起增加了re 模块，它提供 Perl 风格的正则表达式模式。

re 模块使 Python 语言拥有全部的正则表达式功能。

compile 函数根据一个模式字符串和可选的标志参数生成一个正则表达式对象。该对象拥有一系列方法用于正则表达式匹配和替换。

re 模块也提供了与这些方法功能完全一致的函数，这些函数使用一个模式字符串做为它们的第一个参数。

本章节主要介绍Python中常用的正则表达式处理函数。

re.match函数

re.match 尝试从字符串的开始匹配一个模式。

函数语法：

```
re.match(pattern, string, flags=0)
```

函数参数说明：

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

匹配成功re.match方法返回一个匹配的对象，否则返回None。

我们可以使用group(num) 或 groups() 匹配对象函数来获取匹配表达式。

匹配对象方法	描述
group(num=0)	匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。
groups()	返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。

实例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

以上实例执行结果如下：

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

re.search方法

re.match 尝试从字符串的开始匹配一个模式。

函数语法：

```
re.search(pattern, string, flags=0)
```

函数参数说明：

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

匹配成功re.search方法方法返回一个匹配的对象，否则返回None。

我们可以使用group(num) 或 groups() 匹配对象函数来获取匹配表达式。

匹配对象方法	描述
group(num=0)	匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。
groups()	返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。

实例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

以上实例执行结果如下：

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

re.match与re.search的区别

re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

实例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print "match --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"

matchObj = re.search( r'dogs', line, re.M|re.I)
if matchObj:
   print "search --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"
```

以上实例运行结果如下：

```
No match!!
search --> matchObj.group() :  dogs
```

检索和替换

Python 的re模块提供了re.sub用于替换字符串中的匹配项。

语法：

```
re.sub(pattern, repl, string, max=0)
```

返回的字符串是在字符串中用 RE 最左边不重复的匹配来替换。如果模式没有发现，字符将被没有改变地返回。

可选参数 count 是模式匹配后替换的最大次数；count 必须是非负整数。缺省值是 0 表示替换所有的匹配。

实例：

```
#!/usr/bin/python
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*/o>, "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

以上实例执行结果如下：

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

正则表达式修饰符 - 可选标志

正则表达式可以包含一些可选标志修饰符来控制匹配的模式。修饰符被指定为一个可选的标志。多个标志可以通过按位 OR(|) 它们来指定。如 re.I | re.M 被设置成 I 和 M 标志：

修饰符	描述
re.I	使匹配对大小写不敏感
re.L	做本地化识别（locale-aware）匹配
re.M	多行匹配，影响 ^ 和 $
re.S	使 . 匹配包括换行在内的所有字符
re.U	根据Unicode字符集解析字符。这个标志影响 \w, \W, \b, \B.
re.X	该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。

正则表达式模式

模式字符串使用特殊的语法来表示一个正则表达式：

字母和数字表示他们自身。一个正则表达式模式中的字母和数字匹配同样的字符串。

多数字母和数字前加一个反斜杠时会拥有不同的含义。

标点符号只有被转义时才匹配自身，否则它们表示特殊的含义。

反斜杠本身需要使用反斜杠转义。

由于正则表达式通常都包含反斜杠，所以你最好使用原始字符串来表示它们。模式元素(如 r'/t'，等价于'//t')匹配相应的特殊字符。

下表列出了正则表达式模式语法中的特殊元素。如果你使用模式的同时提供了可选的标志参数，某些模式元素的含义会改变。

模式	描述
^	匹配字符串的开头
$	匹配字符串的末尾。
.	匹配任意字符，除了换行符，当re.DOTALL标记被指定时，则可以匹配包括换行符的任意字符。
[...]	用来表示一组字符,单独列出：[amk] 匹配 'a'，'m'或'k'
...	不在[]中的字符：abc 匹配除了a,b,c之外的字符。
re*	匹配0个或多个的表达式。
re+	匹配1个或多个的表达式。
re?	匹配0个或1个由前面的正则表达式定义的片段，贪婪方式
re{ n}	
re{ n,}	精确匹配n个前面表达式。
re{ n, m}	匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式
a | b	匹配a或b
(re)	G匹配括号内的表达式，也表示一个组
(?imx)	正则表达式包含三种可选标志：i, m, 或 x 。只影响括号中的区域。
(?-imx)	正则表达式关闭 i, m, 或 x 可选标志。只影响括号中的区域。
(?: re)	类似 (...), 但是不表示一个组
(?imx: re)	在括号中使用i, m, 或 x 可选标志

(?-imx: re)	在括号中不使用i, m, 或 x 可选标志
(?#...)	注释.
(?= re)	前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。
(?! re)	前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功
(?> re)	匹配的独立模式，省去回溯。
\w	匹配字母数字
\W	匹配非字母数字
\s	匹配任意空白字符，等价于 [\t\n\r\f].
\S	匹配任意非空字符
\d	匹配任意数字，等价于 [0-9].
\D	匹配任意非数字
\A	匹配字符串开始
\Z	匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。c
\z	匹配字符串结束
\G	匹配最后匹配完成的位置。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
\n, \t, 等.	匹配一个换行符。匹配一个制表符。等
\1...\9	比赛第n个分组的子表达式。
\10	匹配第n个分组的子表达式，如果它经匹配。否则指的是八进制字符码的表达式。

正则表达式实例

字符匹配

实例	描述
python	匹配 "python".

字符类

实例	描述
[Pp]ython	匹配 "Python" 或 "python"
rub[ye]	匹配 "ruby" 或 "rube"
[aeiou]	匹配中括号内的任意一个字母
[0-9]	匹配任何数字。类似于 [0123456789]
[a-z]	匹配任何小写字母
[A-Z]	匹配任何大写字母
[a-zA-Z0-9]	匹配任何字母及数字
aeiou	除了aeiou字母以外的所有字符
0-9	匹配除了数字外的字符

特殊字符类

实例	描述
.	匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。
\d	匹配一个数字字符。等价于 [0-9]。
\D	匹配一个非数字字符。等价于 0-9。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 \f\n\r\t\v。
\w	匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9_]'。
\W	匹配任何非单词字符。等价于 'A-Za-z0-9_'。

Python CGI编程

什么是CGI

CGI 目前由NCSA维护，NCSA定义CGI如下：

CGI(Common Gateway Interface),通用网关接口,它是一段程序,运行在服务器上如：HTTP服务器，提供同客户端HTML页面的接口。

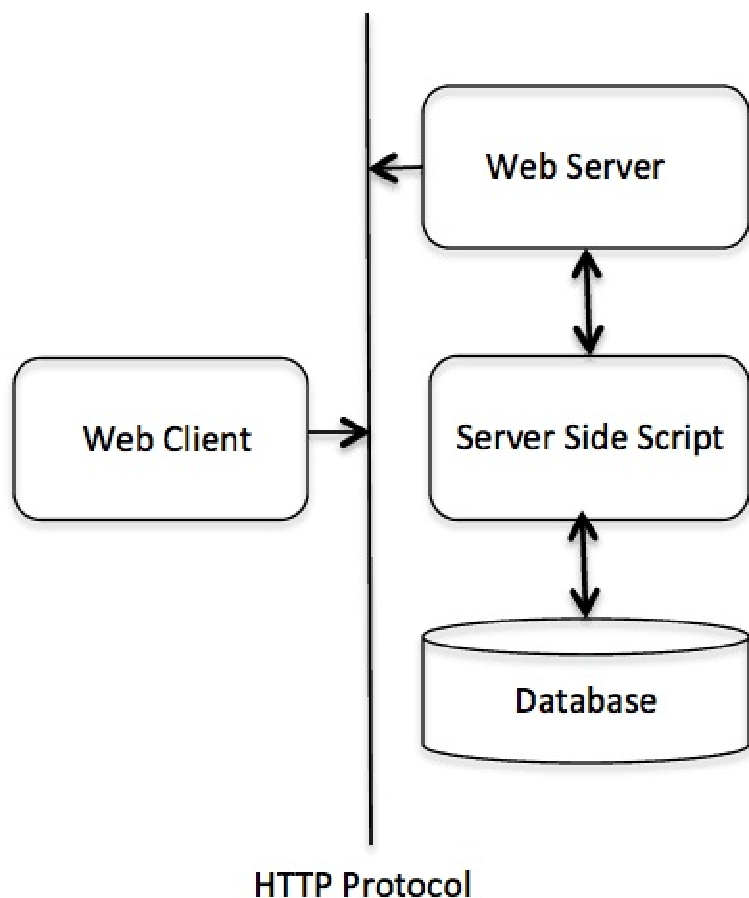
网页浏览

为了更好的了解CGI是如何工作的，我们可以从在网页上点击一个链接或URL的流程：

- 1、使用你的浏览器访问URL并连接到HTTP web 服务器。
- 2、Web服务器接收到请求信息后会解析URL，并查找访问的文件在服务器上是否存在，如果存在返回文件的内容，否则返回错误信息。
- 3、浏览器从服务器上接收信息，并显示接收的文件或者错误信息。

CGI程序可以是Python脚本，PERL脚本，SHELL脚本，C或者C++程序等。

CGI架构图



Web服务器支持及配置

在你进行CGI编程前，确保您的Web服务器支持CGI及已经配置了CGI的处理程序。

所有的HTTP服务器执行CGI程序都保存在一个预先配置的目录。这个目录被称为CGI目录，并按照惯例，它被命名为/var/www/cgi-bin目录。

CGI文件的扩展名为.cgi，python也可以使用.py扩展名。

默认情况下，Linux服务器配置运行的cgi-bin目录中为/var/www。

如果你想指定其他运行CGI脚本的目录，可以修改httpd.conf配置文件，如下所示：

```
<Directory "/var/www/cgi-bin">
   AllowOverride None
   Options ExecCGI
   Order allow,deny
   Allow from all
</Directory>

<Directory "/var/www/cgi-bin">
Options All
</Directory>
```

第一个CGI程序

我们使用Python创建第一个CGI程序，文件名为hellp.py，文件位于/var/www/cgi-bin目录中，内容如下，修改文件的权限为755：

```
#!/usr/bin/python

print "Content-type:text/html\r\n\r\n"
print '<html>'
print '<head>'
print '<title>Hello Word - First CGI Program</title>'
print '</head>'
print '<body>'
print '<h2>Hello Word! This is my first CGI program</h2>'
print '</body>'
print '</html>'
```

以上程序在浏览器访问显示结果如下：

```
Hello Word! This is my first CGI program
```

这个的hello.py脚本是一个简单的Python脚本，脚本第一的输出内容"Content-type:text/html\r\n\r\n"发送到浏览器并告知浏览器显示的内容类型为"text/html"。

HTTP头部

hello.py文件内容中的" Content-type:text/html\r\n\r\n"即为HTTP头部的一部分，它会发送给浏览器告诉浏览器文件的内容类型。

HTTP头部的格式如下：

```
HTTP 字段名: 字段内容

例如
Content-type: text/html\r\n\r\n
```

以下表格介绍了CGI程序中HTTP头部经常使用的信息：

头	描述
Content-type:	请求的与实体对应的MIME信息。例如: Content-type:text/html
Expires: Date	响应过期的日期和时间
Location: URL	用来重定向接收方到非请求URL的位置来完成请求或标识新的资源
Last-modified: Date	请求资源的最后修改时间
Content-length: N	请求的内容长度
Set-Cookie: String	设置Http Cookie

CGI环境变量

所有的CGI程序都接收以下的环境变量，这些变量在CGI程序中发挥了重要的作用：

变量名	描述
CONTENT_TYPE	这个环境变量的值指示所传递来的信息的MIME类型。目前，环境变量CONTENT_TYPE一般都是：application/x-www-form-urlencoded,他表示数据来自于HTML表单。
CONTENT_LENGTH	如果服务器与CGI程序信息的传递方式是POST，这个环境变量即使从标准输入STDIN中可以读到的有效数据的字节数。这个环境变量在读取所输入的数据时必须使用。
HTTP_COOKIE	客户机内的 COOKIE 内容。
HTTP_USER_AGENT	提供包含了版本数或其他专有数据的客户浏览器信息。
PATH_INFO	这个环境变量的值表示紧接在CGI程序名之后的其他路径信息。它常常作为CGI程序的参数出现。
QUERY_STRING	如果服务器与CGI程序信息的传递方式是GET，这个环境变量的值即使所传递的信息。这个信息经跟在CGI程序名的后面，两者中间用一个问号'?'分隔。
REMOTE_ADDR	这个环境变量的值是发送请求的客户机的IP地址，例如上面的192.168.1.67。这个值总是存在的。而且它是Web客户机需要提供给Web服务器的唯一标识，可以在CGI程序中用它来区分不同的Web客户机。
REMOTE_HOST	这个环境变量的值包含发送CGI请求的客户机的主机名。如果不支持你想查询，则无需定义此环境变量。
REQUEST_METHOD	提供脚本被调用的方法。对于使用 HTTP/1.0 协议的脚本，仅 GET 和 POST 有意义。
SCRIPT_FILENAME	CGI脚本的完整路径
SCRIPT_NAME	CGI脚本的的名称
SERVER_NAME	这是你的 WEB 服务器的主机名、别名或IP地址。
SERVER_SOFTWARE	这个环境变量的值包含了调用CGI程序的HTTP服务器的名称和版本号。例如，上面的值为Apache/2.2.14(Unix)

以下是一个简单的CGI脚本输出CGI的环境变量：

```
#!/usr/bin/python

import os

print "Content-type: text/html\r\n\r\n";
print "Environment<\br>";
for param in os.environ.keys():
  print "<b>%20s</b>: %s<\br>" % (param, os.environ[param])
```

GET和POST方法

浏览器客户端通过两种方法向服务器传递信息，这两种方法就是 GET 方法和 POST 方法。

使用GET方法传输数据

GET方法发送编码后的用户信息到服务端，数据信息包含在请求页面的URL上，以"?"号分割, 如下所示：

```
http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2
```

有关 GET 请求的其他一些注释：

- GET 请求可被缓存
- GET 请求保留在浏览器历史记录中
- GET 请求可被收藏为书签
- GET 请求不应在处理敏感数据时使用
- GET 请求有长度限制
- GET 请求只应当用于取回数据

简单的url实例：GET方法

以下是一个简单的URL，使用GET方法向hello_get.py程序发送两个参数：

```
/cgi-bin/hello_get.py?first_name=ZARA&last_name=ALI
```

以下为hello_get.py文件的代码：

```
#!/usr/bin/python

# CGI处理模块
import cgi, cgitb

# 创建 FieldStorage 的实例化
form = cgi.FieldStorage()

# 获取数据
first_name = form.getvalue('first_name')
last_name  = form.getvalue('last_name')

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Hello - Second CGI Program</title>"
print "</head>"
print "<body>"
print "<h2>Hello %s %s</h2>" % (first_name, last_name)
print "</body>"
print "</html>"
```

浏览器请求输出结果：

```
Hello ZARA ALI
```

简单的表单实例：GET方法

以下是一个通过HTML的表单使用GET方法向服务器发送两个数据，提交的服务器脚本同样是hello_get.py文件，代码如下：

```
<form action="/cgi-bin/hello_get.py" method="get">
First Name: <input type="text" name="first_name">  <br />

Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
```

使用POST方法传递数据

使用POST方法向服务器传递数据是更安全可靠的，像一些敏感信息如用户密码等需要使用POST传输数据。

以下同样是hello_get.py ，它也可以处理浏览器提交的POST表单数据:

```
#!/usr/bin/python

# 引入 CGI 模块
import cgi, cgitb

# 创建 FieldStorage 实例
form = cgi.FieldStorage()

# 获取表单数据
first_name = form.getvalue('first_name')
last_name  = form.getvalue('last_name')

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Hello - Second CGI Program</title>"
print "</head>"
print "<body>"
print "<h2>Hello %s %s</h2>" % (first_name, last_name)
print "</body>"
print "</html>"
```

以下为表单通过POST方法向服务器脚本hello_get.py提交数据:

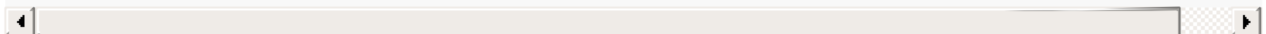
```
<form action="/cgi-bin/hello_get.py" method="post">
First Name: <input type="text" name="first_name"><br />
Last Name: <input type="text" name="last_name" />

<input type="submit" value="Submit" />
</form>
```

通过CGI程序传递checkbox数据

checkbox用于提交一个或者多个选项数据，HTML代码如下：

```
<form action="/cgi-bin/checkbox.cgi" method="POST" target="_blank">
<input type="checkbox" name="maths" value="on" /> Maths
<input type="checkbox" name="physics" value="on" /> Physics
<input type="submit" value="Select Subject" />
</form>
```



以下为 checkbox.cgi 文件的代码：

```
#!/usr/bin/python

# 引入 CGI 处理模块
import cgi, cgitb

# 创建 FieldStorage的实例
form = cgi.FieldStorage()

# 接收字段数据
if form.getvalue('maths'):
   math_flag = "ON"
else:
   math_flag = "OFF"

if form.getvalue('physics'):
   physics_flag = "ON"
else:
   physics_flag = "OFF"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Checkbox - Third CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> CheckBox Maths is : %s</h2>" % math_flag
print "<h2> CheckBox Physics is : %s</h2>" % physics_flag
print "</body>"
print "</html>"
```

通过CGI程序传递Radio数据

Radio只向服务器传递一个数据，HTML代码如下：

```
<form action="/cgi-bin/radiobutton.py" method="post" target="_blank
<input type="radio" name="subject" value="maths" /> Maths
<input type="radio" name="subject" value="physics" /> Physics
<input type="submit" value="Select Subject" />
</form>
```

radiobutton.py 脚本代码如下：

```
#!/usr/bin/python

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

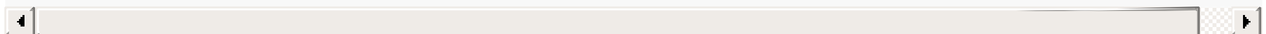
# Get data from fields
if form.getvalue('subject'):
   subject = form.getvalue('subject')
else:
   subject = "Not set"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Radio - Fourth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Selected Subject is %s</h2>" % subject
print "</body>"
print "</html>"
```

通过CGI程序传递 Textarea 数据

Textarea向服务器传递多行数据，HTML代码如下：

```
<form action="/cgi-bin/textarea.py" method="post" target="_blank">
<textarea name="textcontent" cols="40" rows="4">
Type your text here...
</textarea>
<input type="submit" value="Submit" />
</form>
```



textarea.cgi脚本代码如下：

```
#!/usr/bin/python

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('textcontent'):
   text_content = form.getvalue('textcontent')
else:
   text_content = "Not entered"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>";
print "<title>Text Area - Fifth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Entered Text Content is %s</h2>" % text_content
print "</body>"
```

通过CGI程序传递下拉数据

HTML下拉框代码如下：

```
<form action="/cgi-bin/dropdown.py" method="post" target="_blank">
<select name="dropdown">
<option value="Maths" selected>Maths</option>
<option value="Physics">Physics</option>
</select>
<input type="submit" value="Submit"/>
</form>
```

dropdown.py 脚本代码如下所示：

```
#!/usr/bin/python

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('dropdown'):
   subject = form.getvalue('dropdown')
else:
   subject = "Not entered"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Dropdown Box - Sixth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Selected Subject is %s</h2>" % subject
print "</body>"
print "</html>"
```

CGI中使用Cookie

在http协议一个很大的缺点就是不作用户身份的判断，这样给编程人员带来很大的不便，

而cookie功能的出现弥补了这个缺憾。

所有cookie就是在客户访问脚本的同时，通过客户的浏览器，在客户硬盘上写入纪录数据，当下次客户访问脚本时取回数据信息，从而达到身份判别的功能，cookie常用在密码判断中 。

cookie的语法

http cookie的发送是通过http头部来实现的，他早于文件的传递，头部set-cookie的语法如下：

```
Set-cookie:name=name;expires=date;path=path;domain=domain;secure
```

- name=name: 需要设置cookie的值(name不能使用"；"和"，"号),有多个name值时用"；"分隔例如：name1=name1;name2=name2;name3=name3。
- expires=date: cookie的有效期限,格式： expires="Wdy,DD-Mon-YYYY HH:MM:SS"

- path=path: 设置cookie支持的路径,如果path是一个路径，则cookie对这个目录下的所有文件及子目录生效，例如： path="/cgi-bin/"，如果path是一个文件，则cookie指对这个文件生效，例如：path="/cgi-bin/cookie.cgi"。
- domain=domain: 对cookie生效的域名，例如：domain="www.chinalb.com"
- secure: 如果给出此标志，表示cookie只能通过SSL协议的https服务器来传递。
- cookie的接收是通过设置环境变量HTTP_COOKIE来实现的，CGI程序可以通过检索该变量获取cookie信息。

Cookie设置

Cookie的设置非常简单，cookie会在http头部单独发送。以下实例在cookie中设置了UserID 和 Password：

```
<pre>
#!/usr/bin/python

print "Set-Cookie:UserID=XYZ;\r\n"
print "Set-Cookie:Password=XYZ123;\r\n"
print "Set-Cookie:Expires=Tuesday, 31-Dec-2007 23:12:40 GMT";\r\n"
print "Set-Cookie:Domain=www.w3cschool.cc;\r\n"
print "Set-Cookie:Path=/perl;\n"
print "Content-type:text/html\r\n\r\n"
...........Rest of the HTML Content....
```

以上实例使用了 Set-Cookie 头信息来设置Cookie信息，可选项中设置了Cookie的其他属性，如过期时间Expires，域名Domain，路径Path。这些信息设置在"Content-type:text/html\r\n\r\n"之前。

检索Cookie信息

Cookie信息检索页非常简单，Cookie信息存储在CGI的环境变量HTTP_COOKIE中，存储格式如下：

```
key1=value1;key2=value2;key3=value3....
```

以下是一个简单的CGI检索cookie信息的程序：

```
#!/usr/bin/python

# Import modules for CGI handling
from os import environ
import cgi, cgitb

if environ.has_key('HTTP_COOKIE'):
   for cookie in map(strip, split(environ['HTTP_COOKIE'], ';')):
      (key, value ) = split(cookie, '=');
      if key == "UserID":
         user_id = value

      if key == "Password":
         password = value

print "User ID  = %s" % user_id
print "Password = %s" % password
```

以上脚本输出结果如下：

```
User ID = XYZ
Password = XYZ123
```

文件上传实例：

HTML设置上传文件的表单需要设置enctype 属性为multipart/form-data，代码如下所示：

```
<html>
<body>
   <form enctype="multipart/form-data"
                     action="save_file.py" method="post">
   <p>File: <input type="file" name="filename" /></p>
   <p><input type="submit" value="Upload" /></p>
   </form>
</body>
</html>
```

save_file.py脚本文件代码如下：

```
#!/usr/bin/python

import cgi, os
import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# 获取文件名
fileitem = form['filename']

# 检测文件是否上传
if fileitem.filename:
   # 设置文件路径
   fn = os.path.basename(fileitem.filename)
   open('/tmp/' + fn, 'wb').write(fileitem.file.read())

   message = 'The file "' + fn + '" was uploaded successfully'

else:
   message = 'No file was uploaded'

print """\
Content-Type: text/html\n
<html>
<body>
   <p>%s</p>
</body>
</html>
""" % (message,)
```

如果你使用的系统是Unix/Linux，你必须替换文件分隔符，在window下只需要使用open()语句即可：

```
fn = os.path.basename(fileitem.filename.replace("\\", "/" ))
```

文件下载对话框

如果我们需要为用户提供文件下载链接，并在用户点击链接后弹出文件下载对话框，我们通过设置HTTP头信息来实现这些功能，功能代码如下：

```
#!/usr/bin/python

# HTTP Header
print "Content-Type:application/octet-stream; name=\"FileName\"\r\n
print "Content-Disposition: attachment; filename=\"FileName\"\r\n\r

# Actual File Content will go hear.
fo = open("foo.txt", "rb")

str = fo.read();
print str

# Close opend file
fo.close()
```

python操作mysql数据库

Python 标准数据库接口为 Python DB-API，Python DB-API为开发人员提供了数据库应用编程接口。

Python 数据库接口支持非常多的数据库，你可以选择适合你项目的数据库：

- GadFly
- mSQL
- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase

你可以访问Python数据库接口及API查看详细的支持数据库列表。

不同的数据库你需要下载不同的DB API模块，例如你需要访问Oracle数据库和Mysql数据，你需要下载Oracle和MySQL数据库模块。

DB-API 是一个规范. 它定义了一系列必须的对象和数据库存取方式, 以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口 。

Python的DB-API，为大多数的数据库实现了接口，使用它连接各数据库后，就可以用相同的方式操作各数据库。

Python DB-API使用流程：

- 引入 API 模块。
- 获取与数据库的连接。
- 执行SQL语句和存储过程。

- 关闭数据库连接。

什么是MySQLdb?

MySQLdb 是用于Python链接Mysql数据库的接口，它实现了 Python 数据库 API 规范 V2.0，基于 MySQL C API 上建立的。

如何安装MySQLdb?

为了用DB-API编写MySQL脚本，必须确保已经安装了MySQL。复制以下代码，并执行：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb
```

如果执行后的输出结果如下所示，意味着你没有安装 MySQLdb 模块：

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    import MySQLdb
ImportError: No module named MySQLdb
```

安装MySQLdb，请访问 http://sourceforge.net/projects/mysql-python ，(Linux平台可以访问：https://pypi.python.org/pypi/MySQL-python)从这里可选择适合您的平台的安装包，分为预编译的二进制文件和源代码安装包。

如果您选择二进制文件发行版本的话，安装过程基本安装提示即可完成。如果从源代码进行安装的话，则需要切换到MySQLdb发行版本的顶级目录，并键入下列命令:

```
$ gunzip MySQL-python-1.2.2.tar.gz
$ tar -xvf MySQL-python-1.2.2.tar
$ cd MySQL-python-1.2.2
$ python setup.py build
$ python setup.py install
```

注意：请确保您有root权限来安装上述模块。

数据库连接

连接数据库前，请先确认以下事项：

- 您已经创建了数据库 TESTDB.
- 在TESTDB数据库中您已经创建了表 EMPLOYEE
- EMPLOYEE表字段为 FIRST_NAME, LAST_NAME, AGE, SEX 和 INCOME。
- 连接数据库TESTDB使用的用户名为 "testuser" ， 密码为 "test123",你可以可以自己设定或者直接使用root用户名及其密码，Mysql数据库用户授权请使用Grant命令。
- 在你的机子上已经安装了 Python MySQLdb 模块。
- 如果您对sql语句不熟悉，可以访问我们的 SQL基础教程

实例：

以下实例链接Mysql的TESTDB数据库：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# 使用execute方法执行SQL语句
cursor.execute("SELECT VERSION()")

# 使用 fetchone() 方法获取一条数据库。
data = cursor.fetchone()

print "Database version : %s " % data

# 关闭数据库连接
db.close()
```

执行以上脚本输出结果如下：

```
Database version : 5.0.45
```

创建数据库表

如果数据库连接存在我们可以使用execute()方法来为数据库创建表，如下所示创建表EMPLOYEE：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# 如果数据表已经存在使用 execute() 方法删除表。
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# 创建数据表SQL语句
sql = """CREATE TABLE EMPLOYEE (
         FIRST_NAME  CHAR(20) NOT NULL,
         LAST_NAME  CHAR(20),
         AGE INT,
         SEX CHAR(1),
         INCOME FLOAT )"""

cursor.execute(sql)

# 关闭数据库连接
db.close()
```

数据库插入操作

以下实例使用执行 SQL INSERT 语句向表 EMPLOYEE 插入记录：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   # 执行sql语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# 关闭数据库连接
db.close()
```

以上例子也可以写成如下形式：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
       ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # 执行sql语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()

# 关闭数据库连接
db.close()
```

实例：

以下代码使用变量向SQL语句中传递参数:

```
..................................
user_id = "test123"
password = "password"

con.execute('insert into Login values("%s", "%s")' % \
             (user_id, password))
..................................
```

数据库查询操作

Python查询Mysql使用 fetchone() 方法获取单条数据, 使用fetchall() 方法获取多条数据。

- **fetchone():** 该方法获取下一个查询结果集。结果集是一个对象
- **fetchall():**接收全部的返回结果行.

- **rowcount:** 这是一个只读属性，并返回执行execute()方法后影响的行数。

实例：

查询EMPLOYEE表中salary（工资）字段大于1000的所有数据：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 查询语句
sql = "SELECT * FROM EMPLOYEE \
       WHERE INCOME > '%d'" % (1000)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 获取所有记录列表
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
      # 打印结果
      print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
             (fname, lname, age, sex, income )
except:
   print "Error: unable to fecth data"

# 关闭数据库连接
db.close()
```

以上脚本执行结果如下：

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

数据库更新操作

更新操作用于更新数据表的的数据，以下实例将 TESTDB表中的 SEX 字段全部修改为 'M'，AGE 字段递增1：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 更新语句
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1
                          WHERE SEX = '%c'" % ('M')
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()

# 关闭数据库连接
db.close()
```

执行事务

事务机制可以确保数据一致性。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

- 原子性（atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- 一致性（consistency）。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- 隔离性（isolation）。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性（durability）。持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

Python DB API 2.0 的事务提供了两个方法 commit 或 rollback。

实例：

```
# SQL删除记录语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 向数据库提交
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()
```

对于支持事务的数据库， 在Python数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。

commit()方法游标的所有更新操作，rollback（）方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

错误处理

DB API中定义了一些数据库操作的错误及异常，下表列出了这些错误和异常:

异常	描述
Warning	当有严重警告时触发，例如插入数据是被截断等等。必须是 StandardError 的子类。
Error	警告以外所有其他错误类。必须是 StandardError 的子类。
InterfaceError	当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发。 必须是Error的子类。
DatabaseError	和数据库有关的错误发生时触发。 必须是Error的子类。
DataError	当有数据处理时的错误发生时触发，例如：除零错误，数据超范围等等。 必须是DatabaseError的子类。
OperationalError	指非用户控制的，而是操作数据库时发生的错误。例如：连接意外断开、 数据库名未找到、事务处理失败、内存分配错误等等操作数据库是发生的错误。 必须是DatabaseError的子类。
IntegrityError	完整性相关的错误，例如外键检查失败等。必须是DatabaseError子类。
InternalError	数据库的内部错误，例如游标（cursor）失效了、事务同步失败等等。 必须是DatabaseError子类。
ProgrammingError	程序错误，例如数据表（table）没找到或已存在、SQL语句语法错误、 参数数量错误等等。必须是DatabaseError的子类。
NotSupportedError	不支持错误，指使用了数据库不支持的函数或API等。例如在连接对象上 使用.rollback()函数，然而数据库并不支持事务或者事务已关闭。 必须是DatabaseError的子类。

Python使用SMTP发送邮件

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。

python的smtplib提供了一种很方便的途径发送电子邮件。它对smtp协议进行了简单的封装。

Python创建 SMTP 对象语法如下：

```
import smtplib

smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

参数说明：

- host: SMTP 服务器主机。 你可以指定主机的ip地址或者域名如:w3cschool.cc，这个是可选参数。
- port: 如果你提供了 host 参数, 你需要指定 SMTP 服务使用的端口号，一般情况下SMTP端口号为25。
- local_hostname: 如果SMTP在你的本机上，你只需要指定服务器地址为 localhost 即可。

Python SMTP对象使用sendmail方法发送邮件，语法如下：

```
SMTP.sendmail(from_addr, to_addrs, msg[, mail_options, rcpt_options
```

参数说明：

- from_addr: 邮件发送者地址。
- to_addrs: 字符串列表，邮件发送地址。
- msg: 发送消息

这里要注意一下第三个参数，msg是字符串，表示邮件。我们知道邮件一般由标题，发信人，收件人，邮件内容，附件等构成，发送邮件的时候，要注意msg的格式。这个格式就是smtp协议中定义的格式。

实例

以下是一个使用Python发送邮件简单的实例：

```
#!/usr/bin/python

import smtplib

sender = 'from@fromdomain.com'
receivers = ['to@todomain.com']

message = """From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
Subject: SMTP e-mail test

This is a test e-mail message.
"""

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
   print "Successfully sent email"
except SMTPException:
   print "Error: unable to send email"
```

使用Python发送HTML格式的邮件

Python发送HTML格式的邮件与发送纯文本消息的邮件不同之处就是将MIMEText中_subtype设置为html。具体代码如下：

```
import smtplib
from email.mime.text import MIMEText
mailto_list=["YYY@YYY.com"]
mail_host="smtp.XXX.com"  #设置服务器
mail_user="XXX"    #用户名
mail_pass="XXXX"   #口令
mail_postfix="XXX.com"  #发件箱的后缀

def send_mail(to_list,sub,content):  #to_list：收件人；sub：主题；cont
    me="hello"+"<"+mail_user+"@"+mail_postfix+">"   #这里的hello可以
    msg = MIMEText(content,_subtype='html',_charset='gb2312')    #
    msg['Subject'] = sub    #设置主题
    msg['From'] = me
    msg['To'] = ";".join(to_list)
    try:
        s = smtplib.SMTP()
        s.connect(mail_host)  #连接smtp服务器
        s.login(mail_user,mail_pass)  #登陆服务器
        s.sendmail(me, to_list, msg.as_string())  #发送邮件
        s.close()
        return True
    except Exception, e:
        print str(e)
        return False
if __name__ == '__main__':
    if send_mail(mailto_list,"hello","<a href='http://www.cnblogs.c
        print "发送成功"
    else:
        print "发送失败"
```

或者你也可以在消息体中指定Content-type为text/html,如下实例:

```
#!/usr/bin/python

import smtplib

message = """From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
MIME-Version: 1.0
Content-type: text/html
Subject: SMTP HTML e-mail test

This is an e-mail message to be sent in HTML format

<b>This is HTML message.</b>
<h1>This is headline.</h1>
"""

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
   print "Successfully sent email"
except SMTPException:
   print "Error: unable to send email"
```

Python发送带附件的邮件

发送带附件的邮件，首先要创建MIMEMultipart()实例，然后构造附件，如果有多个附件，可依次构造，最后利用smtplib.smtp发送。

```
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import smtplib

#创建一个带附件的实例
msg = MIMEMultipart()

#构造附件1
att1 = MIMEText(open('d:\\123.rar', 'rb').read(), 'base64', 'gb2312
att1["Content-Type"] = 'application/octet-stream'
att1["Content-Disposition"] = 'attachment; filename="123.doc"'#这里
msg.attach(att1)

#构造附件2
att2 = MIMEText(open('d:\\123.txt', 'rb').read(), 'base64', 'gb2312
att2["Content-Type"] = 'application/octet-stream'
att2["Content-Disposition"] = 'attachment; filename="123.txt"'
msg.attach(att2)

#加邮件头
msg['to'] = 'YYY@YYY.com'
msg['from'] = 'XXX@XXX.com'
msg['subject'] = 'hello world'
#发送邮件
try:
    server = smtplib.SMTP()
    server.connect('smtp.XXX.com')
    server.login('XXX','XXXXX')#XXX为用户名，XXXXX为密码
    server.sendmail(msg['from'], msg['to'],msg.as_string())
    server.quit()
    print '发送成功'
except Exception, e:
    print str(e)
```

以下实例指定了Content-type header 为 multipart/mixed，并发送/tmp/test.txt 文本文件：

```
#!/usr/bin/python

import smtplib
import base64

filename = "/tmp/test.txt"

# 读取文件内容并使用 base64 编码
fo = open(filename, "rb")
filecontent = fo.read()
encodedcontent = base64.b64encode(filecontent)  # base64

sender = 'webmaster@tutorialpoint.com'
```

```
reciever = 'amrood.admin@gmail.com'

marker = "AUNIQUEMARKER"

body ="""
This is a test email to send an attachement.
"""
# 定义头部信息
part1 = """From: From Person <me@fromdomain.net>
To: To Person <amrood.admin@gmail.com>
Subject: Sending Attachement
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=%s
--%s
""" % (marker, marker)

# 定义消息动作
part2 = """Content-Type: text/plain
Content-Transfer-Encoding:8bit

%s
--%s
""" % (body,marker)

# 定义附近部分
part3 = """Content-Type: multipart/mixed; name=\"%s\"
Content-Transfer-Encoding:base64
Content-Disposition: attachment; filename=%s

%s
--%s--
""" %(filename, filename, encodedcontent, marker)
message = part1 + part2 + part3

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, reciever, message)
   print "Successfully sent email"
except Exception:
   print "Error: unable to send email"
```

Python多线程

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度
- 程序的运行速度可能加快
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。

线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

每个线程都有他自己的一组CPU寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的CPU寄存器的状态。

指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器，线程总是在进程得到上下文中运行的，这些地址都用于标志拥有线程的进程地址空间中的内存。

- 线程可以被抢占（中断）。
- 在其他线程正在运行时，线程可以暂时搁置（也称为睡眠） -- 这就是线程的退让。

开始学习Python线程

Python中使用线程有两种方式：函数或者用类来包装线程对象。

函数式：调用thread模块中的start_new_thread()函数来产生新线程。语法如下:

```
thread.start_new_thread ( function, args[, kwargs] )
```

参数说明:

- function - 线程函数。
- args - 传递给线程函数的参数,他必须是个tuple类型。
- kwargs - 可选参数。

实例：

```
#!/usr/bin/python

import thread
import time

# 为线程定义一个函数
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print "%s: %s" % ( threadName, time.ctime(time.time()) )

# 创建两个线程
try:
   thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print "Error: unable to start thread"

while 1:
   pass
```

执行以上程序输出结果如下：

```
Thread-1: Thu Jan 22 15:42:17 2009
Thread-1: Thu Jan 22 15:42:19 2009
Thread-2: Thu Jan 22 15:42:19 2009
Thread-1: Thu Jan 22 15:42:21 2009
Thread-2: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:25 2009
Thread-2: Thu Jan 22 15:42:27 2009
Thread-2: Thu Jan 22 15:42:31 2009
Thread-2: Thu Jan 22 15:42:35 2009
```

线程的结束一般依靠线程函数的自然结束；也可以在线程函数中调用thread.exit()，他抛出SystemExit exception，达到退出线程的目的。

线程模块

Python通过两个标准库thread和threading提供对线程的支持。thread提供了低级别的、原始的线程以及一个简单的锁。

thread 模块提供的其他方法：

- threading.currentThread(): 返回当前的线程变量。
- threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程

  启动后、结束前，不包括启动前和终止后的线程。
- threading.activeCount(): 返回正在运行的线程数量，与 len(threading.enumerate())有相同的结果。

除了使用方法外，线程模块同样提供了Thread类来处理线程，Thread类提供了以下方法:

- **run():** 用以表示线程活动的方法。
- **start():**启动线程活动。
- **join([time]):** 等待至线程中止。这阻塞调用线程直至线程的join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- **isAlive():** 返回线程是否活动的。
- **getName():** 返回线程名。
- **setName():** 设置线程名。

使用Threading模块创建线程

使用Threading模块创建线程，直接从threading.Thread继承，然后重写**init**方法和run方法：

```
#!/usr/bin/python

import threading
import time

exitFlag = 0

class myThread (threading.Thread):   #继承父类threading.Thread
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):                   #把要执行的代码写到run函数里面 线程
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启线程
thread1.start()
thread2.start()

print "Exiting Main Thread"
```

以上程序执行结果如下；

```
Starting Thread-1
Starting Thread-2
Exiting Main Thread
Thread-1: Thu Mar 21 09:10:03 2013
Thread-1: Thu Mar 21 09:10:04 2013
Thread-2: Thu Mar 21 09:10:04 2013
Thread-1: Thu Mar 21 09:10:05 2013
Thread-1: Thu Mar 21 09:10:06 2013
Thread-2: Thu Mar 21 09:10:06 2013
Thread-1: Thu Mar 21 09:10:07 2013
Exiting Thread-1
Thread-2: Thu Mar 21 09:10:08 2013
Thread-2: Thu Mar 21 09:10:10 2013
Thread-2: Thu Mar 21 09:10:12 2013
Exiting Thread-2
```

线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。

使用Thread对象的Lock和Rlock可以实现简单的线程同步，这两个对象都有acquire方法和release方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到acquire和release方法之间。如下：

多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。

考虑这样一种情况：一个列表里所有元素都是0，线程"set"从后向前把所有元素改成1，而线程"print"负责从前往后读取列表并打印。

那么，可能线程"set"开始改的时候，线程"print"便来打印列表了，输出就成了一半0一半1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。

锁有两种状态——锁定和未锁定。每当一个线程比如"set"要访问共享数据时，必须先获得锁定；如果已经有别的线程比如"print"获得锁定了，那么就让线程"set"暂停，也就是同步阻塞；等到线程"print"访问完毕，释放锁以后，再让线程"set"继续。

经过这样的处理，打印列表时要么全部输出0，要么全部输出1，不会再出现一半0一半1的尴尬场面。

实例：

```
#!/usr/bin/python

import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
       # 获得锁，成功获得锁定后返回True
       # 可选的timeout参数不填时将一直阻塞直到获得锁定
       # 否则超时后将返回False
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # 释放锁
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

threadLock = threading.Lock()
threads = []

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()

# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)

# 等待所有线程完成
for t in threads:
    t.join()
print "Exiting Main Thread"
```

线程优先级队列（ **Queue**）

Python的Queue模块中提供了同步的、线程安全的队列类，包括FIFO（先入先出)队列Queue，LIFO（后入先出）队列LifoQueue，和优先级队列PriorityQueue。这些队列都实现了锁原语，能够在多线程中直接使用。可以使用队列来实现线程间的同步。

Queue模块中的常用方法:

- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回True,反之False
- Queue.full() 如果队列满了，返回True,反之False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]])获取队列，timeout等待时间
- Queue.get_nowait() 相当Queue.get(False)
- Queue.put(item) 写入队列，timeout等待时间
- Queue.put_nowait(item) 相当Queue.put(item, False)
- Queue.task_done() 在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
- Queue.join() 实际上意味着等到队列为空，再执行别的操作

实例:

```
#!/usr/bin/python

import Queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print "Starting " + self.name
        process_data(self.name, self.q)
        print "Exiting " + self.name

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print "%s processing %s" % (threadName, data)
        else:
            queueLock.release()
        time.sleep(1)
```

```
threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = Queue.Queue(10)
threads = []
threadID = 1

# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1

# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# 等待队列清空
while not workQueue.empty():
    pass

# 通知线程是时候退出
exitFlag = 1

# 等待所有线程完成
for t in threads:
    t.join()
print "Exiting Main Thread"
```

以上程序执行结果：

```
Starting Thread-1
Starting Thread-2
Starting Thread-3
Thread-1 processing One
Thread-2 processing Two
Thread-3 processing Three
Thread-1 processing Four
Thread-2 processing Five
Exiting Thread-3
Exiting Thread-1
Exiting Thread-2
Exiting Main Thread
```

python XML解析

什么是XML？

XML 指可扩展标记语言（**e**X**tensible** **M**arkup **L**anguage）。 你可以通过本站学习XML教程

XML 被设计用来传输和存储数据。

XML是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以标识。

它也是元标记语言，即定义了用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。

python对XML的解析

常见的XML编程接口有DOM和SAX，这两种接口处理XML文件的方式不同，当然使用场合也不同。

python有三种方法解析XML，SAX，DOM，以及ElementTree:

1.SAX (simple API for XML)

pyhton 标准库包含SAX解析器，SAX用事件驱动模型，通过在解析XML的过程中触发一个个的事件并调用用户定义的回调函数来处理XML文件。

2.DOM(Document Object Model)

将XML数据在内存中解析成一个树，通过对树的操作来操作XML。

3.ElementTree(元素树)

ElementTree就像一个轻量级的DOM，具有方便友好的API。代码可用性好，速度快，消耗内存少。

注：因DOM需要将XML数据映射到内存中的树，一是比较慢，二是比较耗内存，而SAX流式读取XML文件，比较快，占用内存少，但需要用户实现回调函数（handler）。

本章节使用到的XML实例文件movies.xml内容如下：

```
<collection shelf="New Arrivals">
<movie title="Enemy Behind">
   <type>War, Thriller</type>
   <format>DVD</format>
   <year>2003</year>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Talk about a US-Japan war</description>
</movie>
<movie title="Transformers">
   <type>Anime, Science Fiction</type>
   <format>DVD</format>
   <year>1989</year>
   <rating>R</rating>
   <stars>8</stars>
   <description>A schientific fiction</description>
</movie>
   <movie title="Trigun">
   <type>Anime, Action</type>
   <format>DVD</format>
   <episodes>4</episodes>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Vash the Stampede!</description>
</movie>
<movie title="Ishtar">
   <type>Comedy</type>
   <format>VHS</format>
   <rating>PG</rating>
   <stars>2</stars>
   <description>Viewable boredom</description>
</movie>
</collection>
```

python使用SAX解析xml

SAX是一种基于事件驱动的API。

利用SAX解析XML文档牵涉到两个部分:解析器和事件处理器。

解析器负责读取XML文档,并向事件处理器发送事件,如元素开始跟元素结束事件;

而事件处理器则负责对事件作出相应,对传递的XML数据进行处理。

- 1、对大型文件进行处理；
- 2、只需要文件的部分内容，或者只需从文件中得到特定信息。
- 3、想建立自己的对象模型的时候。

在python中使用sax方式处理xml要先引入xml.sax中的parse函数，还有xml.sax.handler中的ContentHandler。

ContentHandler类方法介绍

characters(content)方法

调用时机：

从行开始，遇到标签之前，存在字符，content的值为这些字符串。

从一个标签，遇到下一个标签之前， 存在字符，content的值为这些字符串。

从一个标签，遇到行结束符之前，存在字符，content的值为这些字符串。

标签可以是开始标签，也可以是结束标签。

startDocument()方法

文档启动的时候调用。

endDocument()方法

解析器到达文档结尾时调用。

startElement(name, attrs)方法

遇到XML开始标签时调用，name是标签的名字，attrs是标签的属性值字典。

endElement(name)方法

遇到XML结束标签时调用。

make_parser方法

以下方法创建一个新的解析器对象并返回。

```
xml.sax.make_parser( [parser_list] )
```

参数说明:

- **parser_list** - 可选参数，解析器列表

parser方法

以下方法创建一个 SAX 解析器并解析xml文档：

```
xml.sax.parse( xmlfile, contenthandler[, errorhandler])
```

参数说明:

- **xmlfile** - xml文件名
- **contenthandler** - 必须是一个ContentHandler的对象
- **errorhandler** - 如果指定该参数，errorhandler必须是一个SAX ErrorHandler对象

parseString方法

parseString方法创建一个XML解析器并解析xml字符串：

```
xml.sax.parseString(xmlstring, contenthandler[, errorhandler])
```

参数说明:

- **xmlstring** - xml字符串
- **contenthandler** - 必须是一个ContentHandler的对象
- **errorhandler** - 如果指定该参数，errorhandler必须是一个SAX ErrorHandler对象

Python 解析XML实例

```
#!/usr/bin/python

import xml.sax

class MovieHandler( xml.sax.ContentHandler ):
   def __init__(self):
      self.CurrentData = ""
      self.type = ""
      self.format = ""
      self.year = ""
      self.rating = ""
      self.stars = ""
      self.description = ""

   # 元素开始事件处理
   def startElement(self, tag, attributes):
      self.CurrentData = tag
      if tag == "movie":
         print "*****Movie*****"
         title = attributes["title"]
         print "Title:", title

   # 元素结束事件处理
   def endElement(self, tag):
      if self.CurrentData == "type":
         print "Type:", self.type
      elif self.CurrentData == "format":
```

```
            print "Format:", self.format
         elif self.CurrentData == "year":
            print "Year:", self.year
         elif self.CurrentData == "rating":
            print "Rating:", self.rating
         elif self.CurrentData == "stars":
            print "Stars:", self.stars
         elif self.CurrentData == "description":
            print "Description:", self.description
         self.CurrentData = ""

      # 内容事件处理
      def characters(self, content):
         if self.CurrentData == "type":
            self.type = content
         elif self.CurrentData == "format":
            self.format = content
         elif self.CurrentData == "year":
            self.year = content
         elif self.CurrentData == "rating":
            self.rating = content
         elif self.CurrentData == "stars":
            self.stars = content
         elif self.CurrentData == "description":
            self.description = content

   if ( __name__ == "__main__"):

      # 创建一个 XMLReader
      parser = xml.sax.make_parser()
      # turn off namepsaces
      parser.setFeature(xml.sax.handler.feature_namespaces, 0)

      # 重写 ContextHandler
      Handler = MovieHandler()
      parser.setContentHandler( Handler )

      parser.parse("movies.xml")
```

以上代码执行结果如下：

```
*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Year: 2003
Rating: PG
Stars: 10
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Year: 1989
Rating: R
Stars: 8
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Stars: 10
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Stars: 2
Description: Viewable boredom
```

完整的 SAX API 文档请查阅Python SAX APIs

使用xml.dom解析xml

文件对象模型（Document Object Model，简称DOM），是W3C组织推荐的处理可扩展置标语言的标准编程接口。

一个 DOM 的解析器在解析一个 XML 文档时，一次性读取整个文档，把文档中所有元素保存在内存中的一个树结构里，之后你可以利用DOM 提供的不同的函数来读取或修改文档的内容和结构，也可以把修改过的内容写入xml文件。

python中用xml.dom.minidom来解析xml文件，实例如下：

```
#!/usr/bin/python

from xml.dom.minidom import parse
import xml.dom.minidom

# 使用minidom解析器打开 XML 文档
DOMTree = xml.dom.minidom.parse("movies.xml")
collection = DOMTree.documentElement
if collection.hasAttribute("shelf"):
   print "Root element : %s" % collection.getAttribute("shelf")

# 在集合中获取所有电影
movies = collection.getElementsByTagName("movie")

# 打印每部电影的详细信息
for movie in movies:
   print "*****Movie*****"
   if movie.hasAttribute("title"):
      print "Title: %s" % movie.getAttribute("title")

   type = movie.getElementsByTagName('type')[0]
   print "Type: %s" % type.childNodes[0].data
   format = movie.getElementsByTagName('format')[0]
   print "Format: %s" % format.childNodes[0].data
   rating = movie.getElementsByTagName('rating')[0]
   print "Rating: %s" % rating.childNodes[0].data
   description = movie.getElementsByTagName('description')[0]
   print "Description: %s" % description.childNodes[0].data
```

以上程序执行结果如下：

```
Root element : New Arrivals
*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Rating: PG
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Rating: R
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Description: Viewable boredom
```

完整的 DOM API 文档请查阅Python DOM APIs。

python GUI编程(Tkinter)

python提供了多个图形开发界面的库，几个常用Python GUI库如下：

- **Tkinter**： Tkinter模块("Tk 接口")是Python的标准Tk GUI工具包的接口.Tk和Tkinter可以在大多数的Unix平台下使用,同样可以应用在Windows和Macintosh系统里.,Tk8.0的后续版本可以实现本地窗口风格,并良好地运行在绝大多数平台中。
- **wxPython**：wxPython 是一款开源软件，是 Python 语言的一套优秀的 GUI 图形库，允许 Python 程序员很方便的创建完整的、功能键全的 GUI 用户界面。
- **Jython**：Jython程序可以和Java无缝集成。除了一些标准模块，Jython使用Java的模块。Jython几乎拥有标准的Python中不依赖于C语言的全部模块。比如，Jython的用户界面将使用Swing，AWT或者SWT。Jython可以被动态或静态地编译成Java字节码。

Tkinter 编程

Tkinter 是Python的标准GUI库。Python使用Tkinter可以快速的创建GUI应用程序。

由于Tkinter是内置到python的安装包中、只要安装好Python之后就能import Tkinter库、而且IDLE也是用Tkinter编写而成、对于简单的图形界面Tkinter还是能应付自如。

创建一个GUI程序

- 1、导入Tkinter模块
- 2、创建控件
- 3、指定这个控件的master， 即这个控件属于哪一个
- 4、告诉GM(geometry manager)有一个控件产生了。

实例:

```
#!/usr/bin/python

import Tkinter
top = Tkinter.Tk()
# 进入消息循环
top.mainloop()
```

以上代码执行结果如下图:



Tkinter 组件

Tkinter的提供各种控件，如按钮，标签和文本框，一个GUI应用程序中使用。这些控件通常被称为控件或者部件。

目前有15种Tkinter的部件。我们提出这些部件以及一个简短的介绍，在下面的表:

控件	描述
Button	按钮控件；在程序中显示按钮。
Canvas	画布控件；显示图形元素如线条或文本
Checkbutton	多选框控件；用于在程序中提供多项选择框
Entry	输入控件；用于显示简单的文本内容
Frame	框架控件；在屏幕上显示一个矩形区域，多用来作为容器
Label	标签控件；可以显示文本和位图
Listbox	列表框控件；在Listbox窗口小部件是用来显示一个字符串列表给用户
Menubutton	菜单按钮控件，由于显示菜单项。
Menu	菜单控件；显示菜单栏,下拉菜单和弹出菜单
Message	消息控件；用来显示多行文本，与label比较类似
Radiobutton	单选按钮控件；显示一个单选的按钮状态
Scale	范围控件；显示一个数值刻度，为输出限定范围的数字区间
Scrollbar	滚动条控件，当内容超过可视化区域时使用，如列表框。.
Text	文本控件；用于显示多行文本
Toplevel	容器控件；用来提供一个单独的对话框，和Frame比较类似
Spinbox	输入控件；与Entry类似，但是可以指定输入范围值
PanedWindow	PanedWindow是一个窗口布局管理的插件，可以包含一个或者多个子控件。
LabelFrame	labelframe 是一个简单的容器控件。常用与复杂的窗口布局。
tkMessageBox	用于显示你应用程序的消息框。

标准属性

标准属性也就是所有控件的共同属性，如大小，字体和颜色等等。

属性	描述
Dimension	控件大小；
Color	控件颜色；
Font	控件字体；
Anchor	锚点；
Relief	控件样式；
Bitmap	位图；
Cursor	光标；

几何管理

Tkinter控件有特定的几何状态管理方法，管理整个控件区域组织，一下是Tkinter公开的几何管理类：包、网格、位置

几何方法	描述
pack()	包装；
grid()	网格；
place()	位置；

Python2.x与3??.x版本区别

Python的3??.0版本，常被称为Python 3000，或简称Py3k。相对于Python的早期版本，这是一个较大的升级。

为了不带入过多的累赘，Python 3.0在设计的时候没有考虑向下相容。

许多针对早期Python版本设计的程式都无法在Python 3.0上正常执行。

为了照顾现有程式，Python 2.6作为一个过渡版本，基本使用了Python 2.x的语法和库，同时考虑了向Python 3.0的迁移，允许使用部分Python 3.0的语法与函数。

新的Python程式建议使用Python 3.0版本的语法。

除非执行环境无法安装Python 3.0或者程式本身使用了不支援Python 3.0的第三方库。目前不支援Python 3.0的第三方库有Twisted, py2exe, PIL等。

大多数第三方库都正在努力地相容Python 3.0版本。即使无法立即使用Python 3.0，也建议编写相容Python 3.0版本的程式，然后使用Python 2.6, Python 2.7来执行。

主要变化

Python 3.0的变化主要在以下几个方面:

print语句没有了，取而代之的是print()函数。 Python 2.6与Python 2.7部分地支持这种形式的print语法。在Python 2.6与Python 2.7里面，以下三种形式是等价的：

```
print "fish"
print ("fish") #注意print后面有个空格
print("fish") #print()不能带有任何其它参数
```

然而，Python 2.6实际已经支持新的print()语法：

```
from __future__ import print_function
print("fish", "panda", sep=', ')
```

新的str类别表示一个Unicode字串，相当于Python 2.x版本的unicode类别。而位元组序列则用类似b"abc"的语法表示，用bytes类表示，相当于Python 2.x的str类别。

现在两种类别不能再隐式地自动转换，因此在Python 3.x里面"fish"+b"panda"是错误。正确的写法是"fish"+b"panda".decode("utf-8")。 Python 2.6可以自动地将位元组序列识别为Unicode字串，方法是：

```
from __future__ import unicode_literals
print(repr("fish"))
```

除法运算符"/"在Python 3.x内总是返回浮点数。而在Python 2.6内会判断被除数与除数是否是整数。如果是整数会返回整数值，相当于整除;浮点数则返回浮点数值。

为了让Python 2.6统一返回浮点数值，可以：

```
from __future__ import division
print(3/2)
```

- 捕获异常的语法由except exc, var改为except exc as var。使用语法except (exc1, exc2) as var可以同时捕获多种类别的异常。 Python 2.6已经支援这两种语法。
- 集合(set) 的新写法：{1,2,3,4}。注意{}仍然表示空的字典(dict) 。
- 字典推导式(Dictionary comprehensions) {expr1: expr2 for k, v in d}，这个语法等价于

```
result={}
for k, v in d.items():
    result[expr1]=expr2
return result
```

集合推导式(Set Comprehensions) {expr1 for x in stuff}。这个语法等价于：

```
result = set()
for x in stuff:
    result.add(expr1)
return result
```

- 八进制数必须写成0o777，原来的形式0777不能用了；二进制必须写成0b111。新增了一个bin()函数用于将一个整数转换成二进制字串。 Python 2.6已经支援这两种语法。
- dict.keys(), dict.values??(), dict.items(), map(), filter(), range(), zip()不再返回列表，而是迭代器。
- 如果两个物件之间没有定义明确的有意义的顺序。使用<, >, <=, >=比较它们会投掷异常。比如1 < ""在Python 2.6里面会返回True，而在Python 3.0里面会投掷异常。现在cmp(), instance.**cmp**()函数已经被删除。
- 可以注释函数的参数与返回值。此特性可方便IDE对原始码进行更深入的分析。例如给参数增加类别讯息：

```
def sendMail(from_: str, to: str, title: str, body: str) -> bool:
    pass
```

- 合并int与long类型。
- 多个模块被改名（根据PEP8）：

旧的名字	新的名字
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
repr	reprlib

- StringIO模块现在被合并到新的io模组内。 new, md5, gopherlib等模块被删除。 Python 2.6已经支援新的io模组。
- httplib, BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer, Cookie, cookielib被合并到http包内。
- 取消了exec语句，只剩下exec()函数。 Python 2.6已经支援exec()函数。

Python IDE

本文为大家推荐几款款不错的**Python IDE**（集成开发环境），比较推荐PyCharm，当然你可以根据自己的喜好来选择适合自己的 Python IDE。

PyCharm

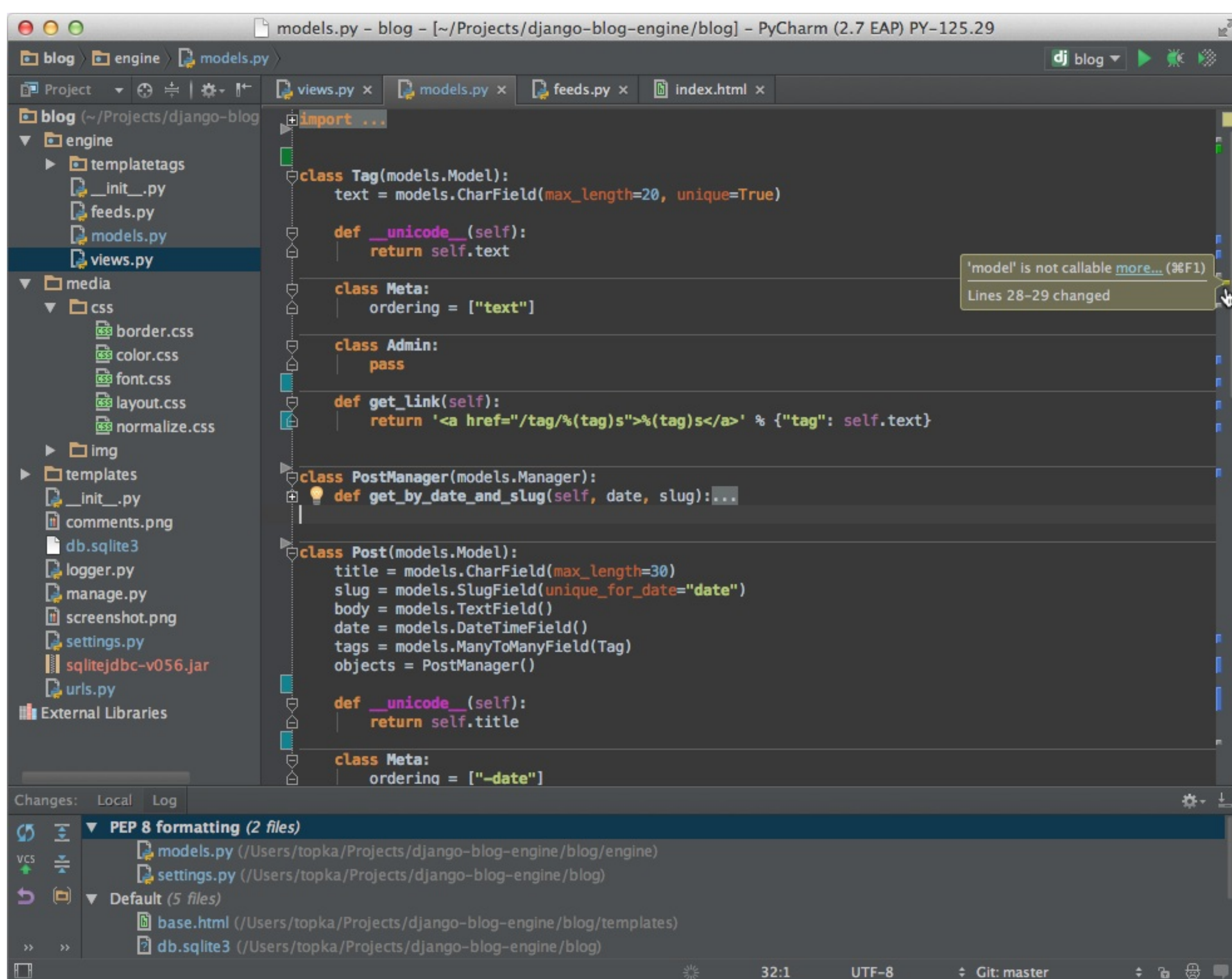
PyCharm是由JetBrains打造的一款Python IDE。

PyCharm具备一般 **Python IDE** 的功能，比如：调试、语法高亮、项目管理、代码跳转、智能提示、自动完成、单元测试、版本控制等。

另外，PyCharm还提供了一些很好的功能用于Django开发，同时支持Google App Engine，更酷的是，PyCharm支持IronPython。

PyCharm 官方下载地址：http://www.jetbrains.com/pycharm/download/

效果图查看：

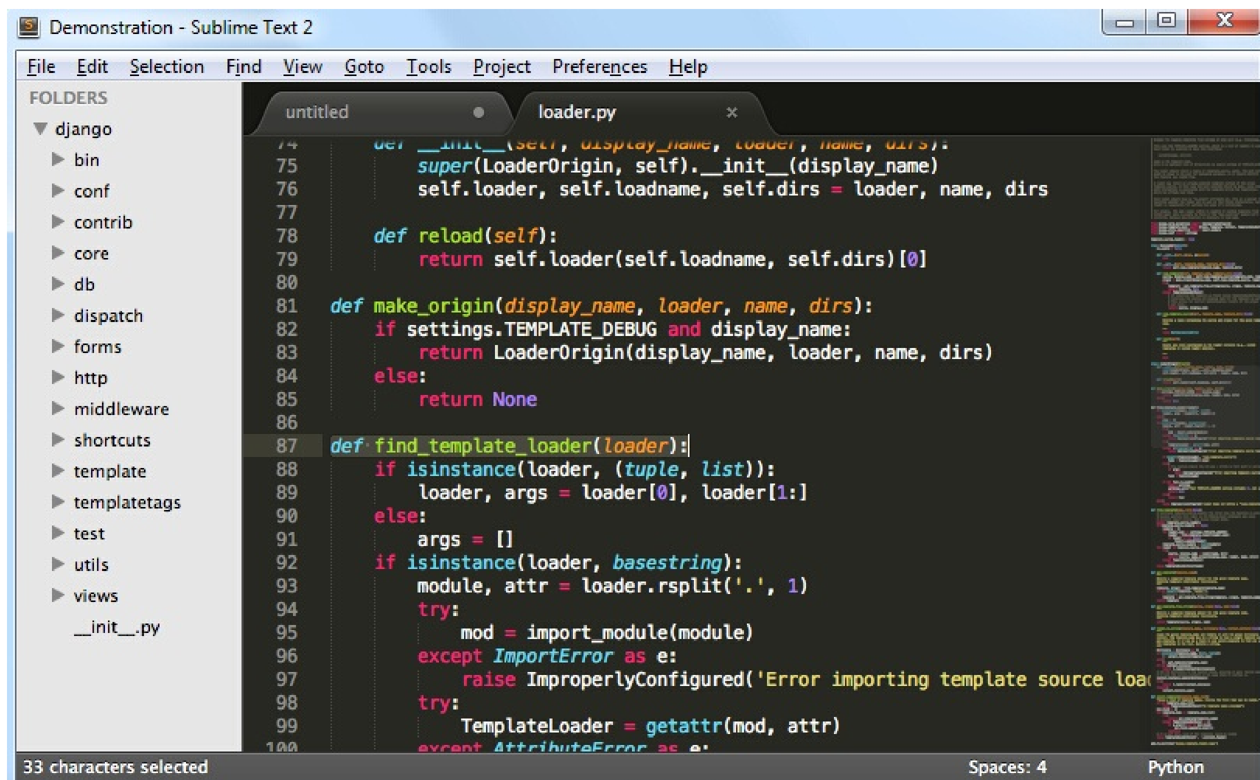


Sublinme Text 2

Sublime Text具有漂亮的用户界面和强大的功能，例如代码缩略图，Python的插件，代码段等。还可自定义键绑定，菜单和工具栏。

Sublime Text 的主要功能包括：拼写检查，书签，完整的 Python API ， Goto 功能，即时项目切换，多选择，多窗口等等。

Sublime Text 是一个跨平台的编辑器，同时支持Windows、Linux、Mac OS X等操作系统。



使用Sublinme Text 2的插件扩展功能，你可以轻松的打造一款不错的Python IDE，以下推荐几款插件（你可以找到更多）：

- CodeIntel：自动补全+成员/方法提示（强烈推荐）
- SublimeREPL：用于运行和调试一些需要交互的程序（E.G. 使用了Input()的程序）
- Bracket Highlighter：括号匹配及高亮
- SublimeLinter：代码pep8格式检查

Eclipse+Pydev

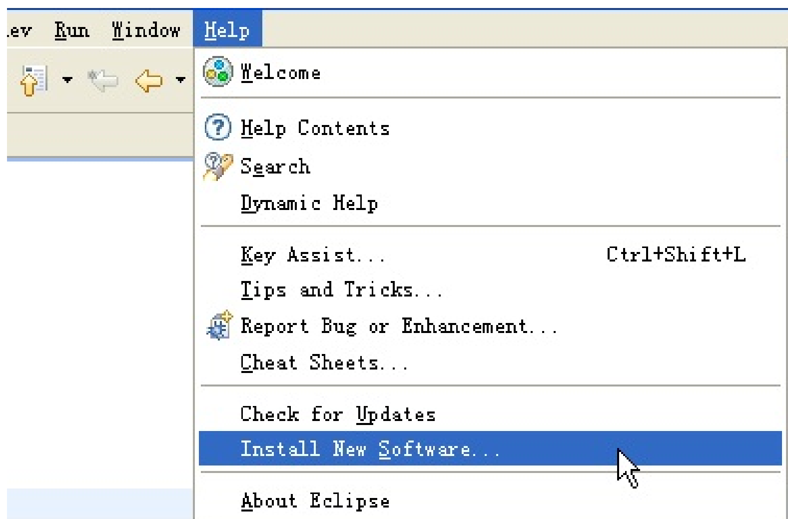
1、安装Eclipse

Eclipse可以在它的官方网站Eclipse.org找到并下载，通常我们可以选择适合自己的Eclipse版本，比如Eclipse Classic。下载完成后解压到到你想安装的目录中即可。

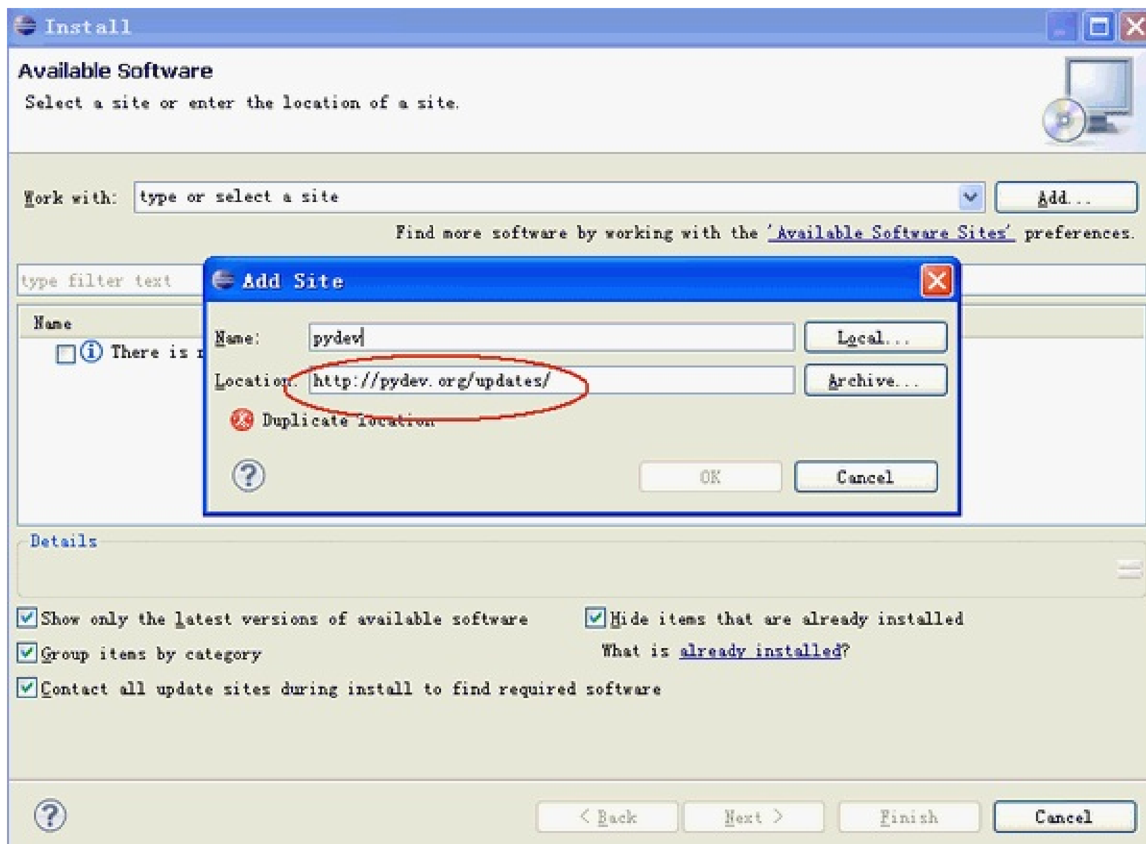
当然在执行Eclipse之前，你必须确认安装了Java运行环境,即必须安装JRE或JDK，你可以到（http://www.java.com/en/download/manual.jsp）找到JRE下载并安装。

2、安装Pydev

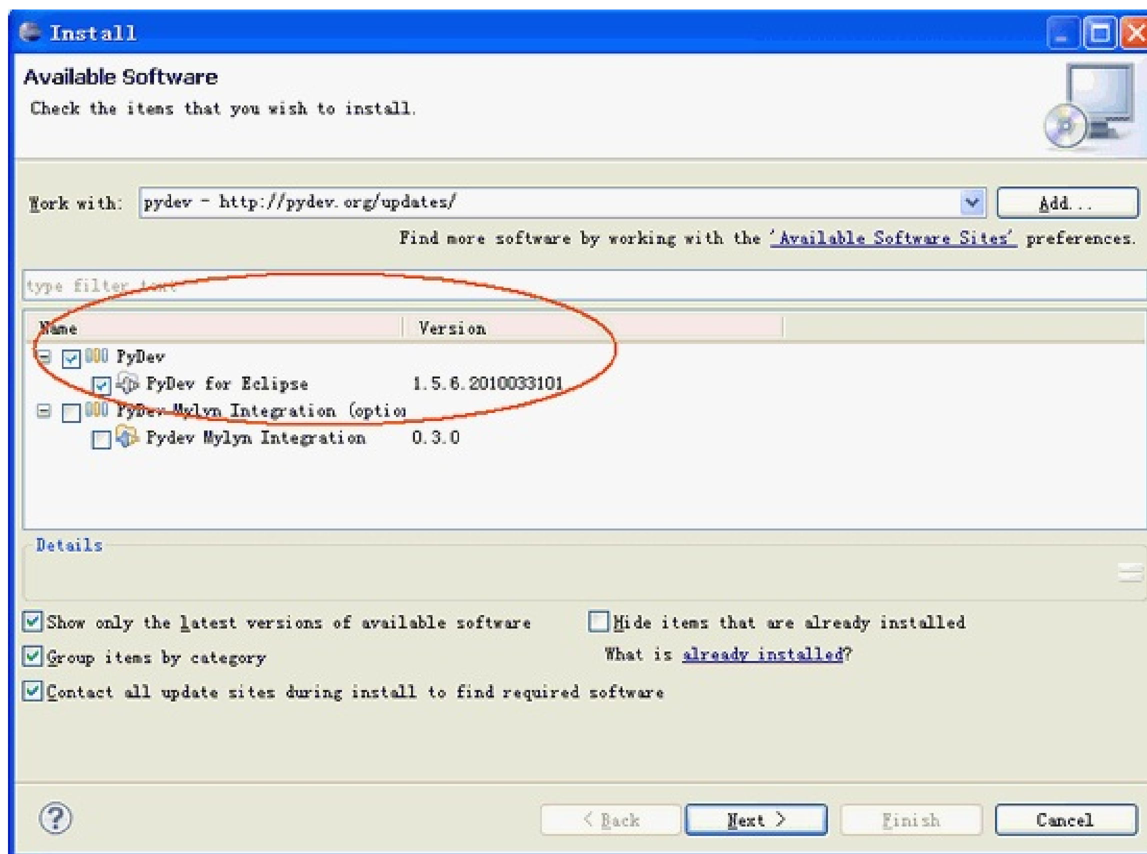
运行Eclipse之后，选择help-->Install new Software，如下图所示。



点击Add，添加pydev的安装地址：http://pydev.org/updates/，如下图所示。



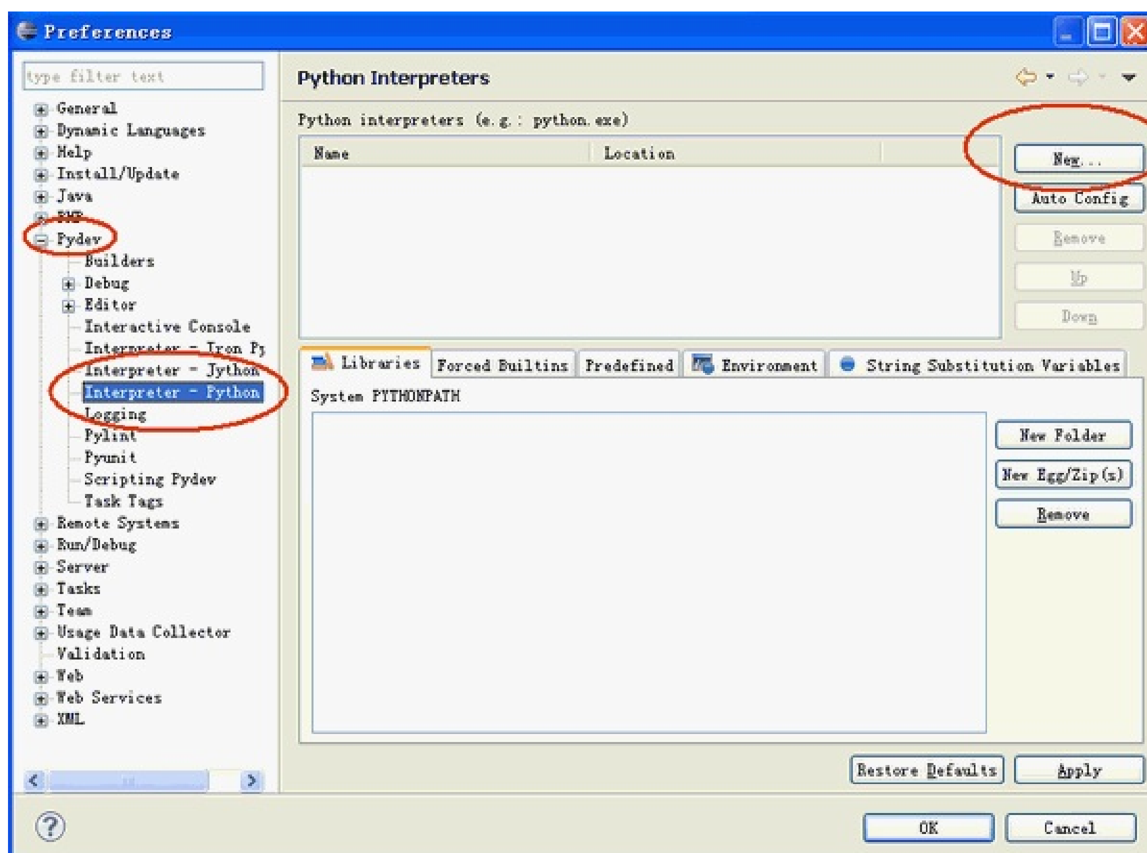
完成后点击"ok"，接着点击PyDev的"+"，展开PyDev的节点，要等一小段时间，让它从网上获取PyDev的相关套件，当完成后会多出PyDev的相关套件在子节点里，勾选它们然后按next进行安装。如下图所示。



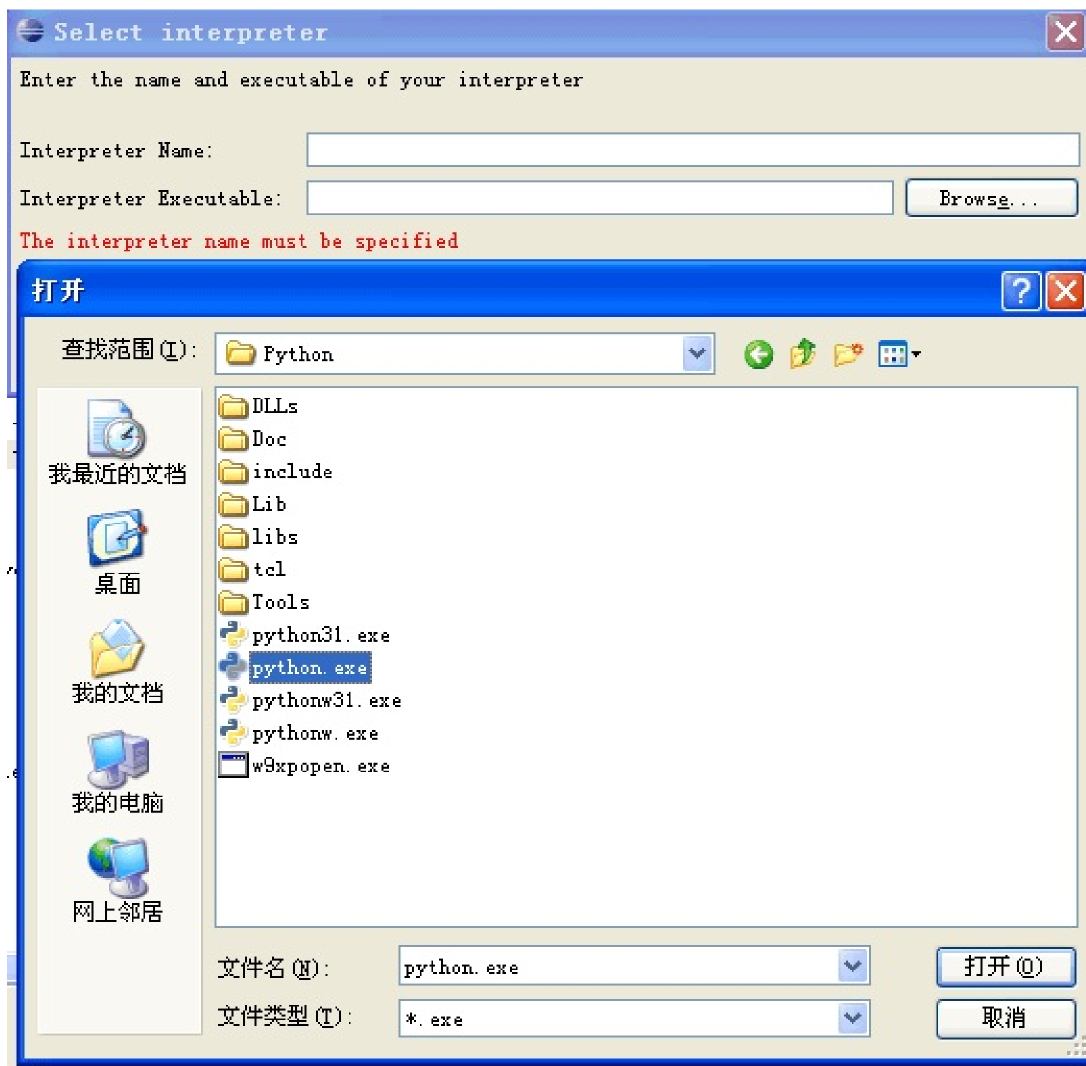
安装完成后，重启Eclipse即可

3、设置Pydev

安装完成后，还需要设置一下PyDev，选择Window -> Preferences来设置PyDev。设置Python的路径，从Pydev的Interpreter - Python页面选择New



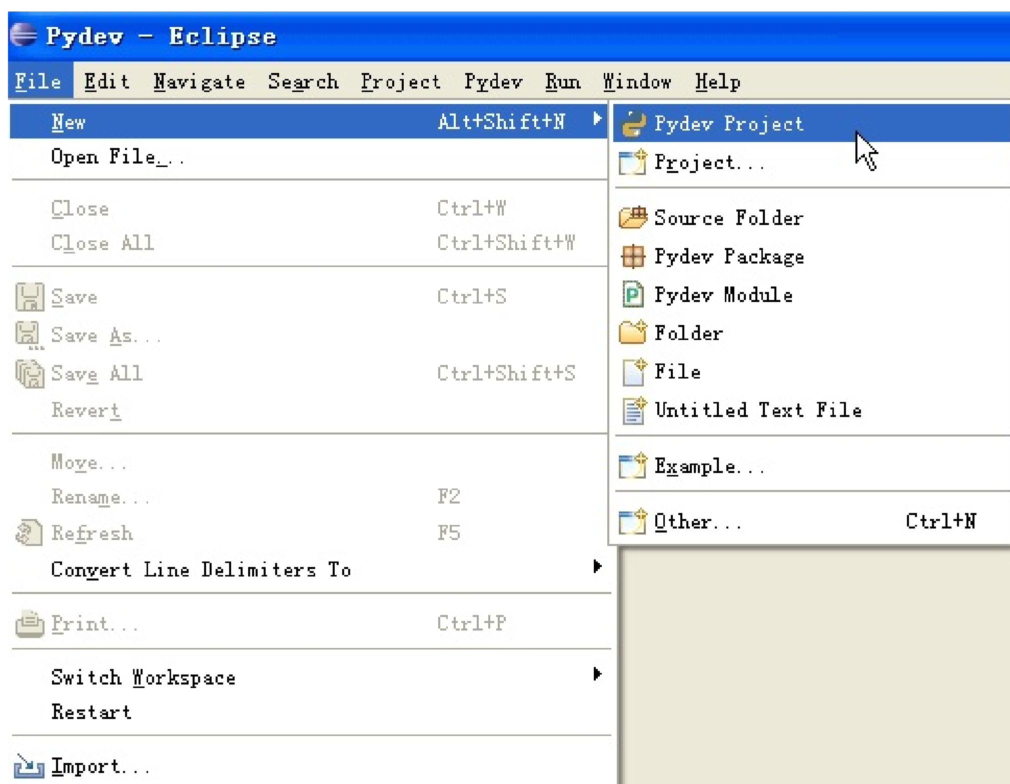
会弹出一个窗口让你选择Python的安装位置，选择你安装Python的所在位置。



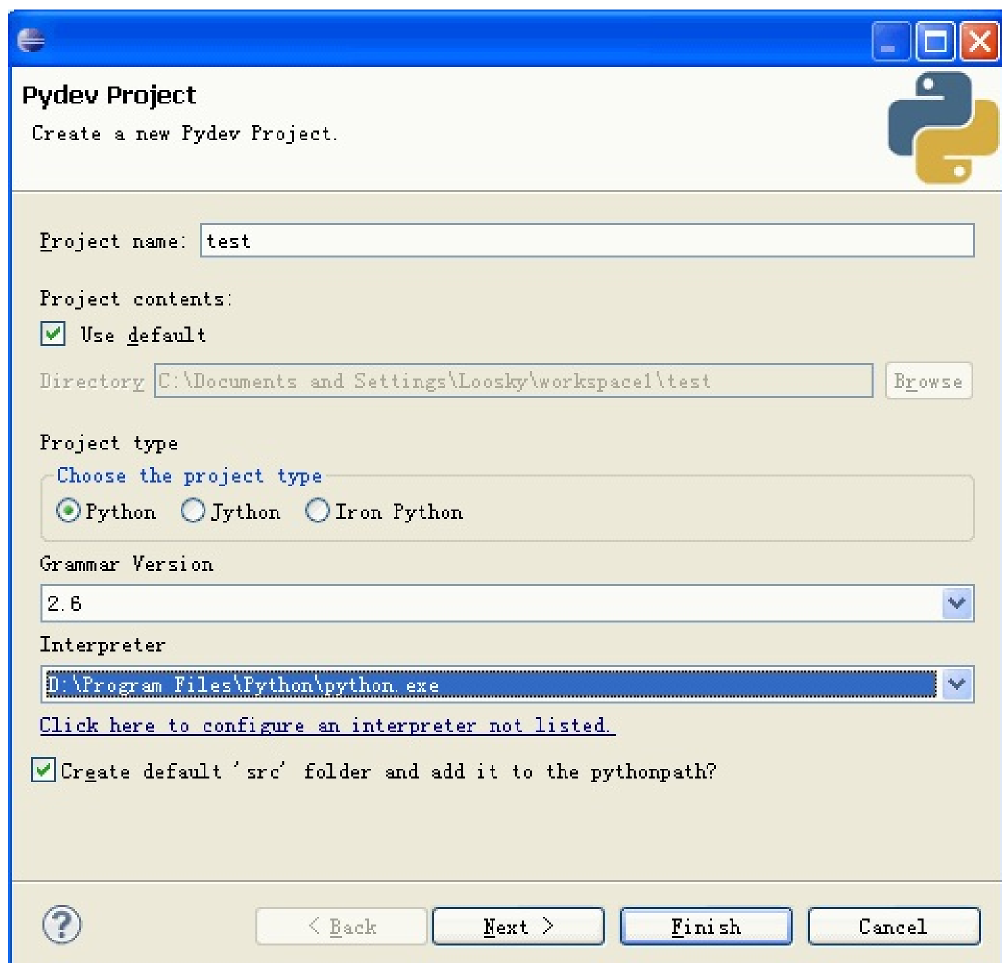
完成之后PyDev就设置完成，可以开始使用。

4、建立Python Project：

安装好Eclipse+PyDev以后，我们就可以开始使用它来开发项目了。首先要创建一个项目，选择File -> New ->Pydev Project

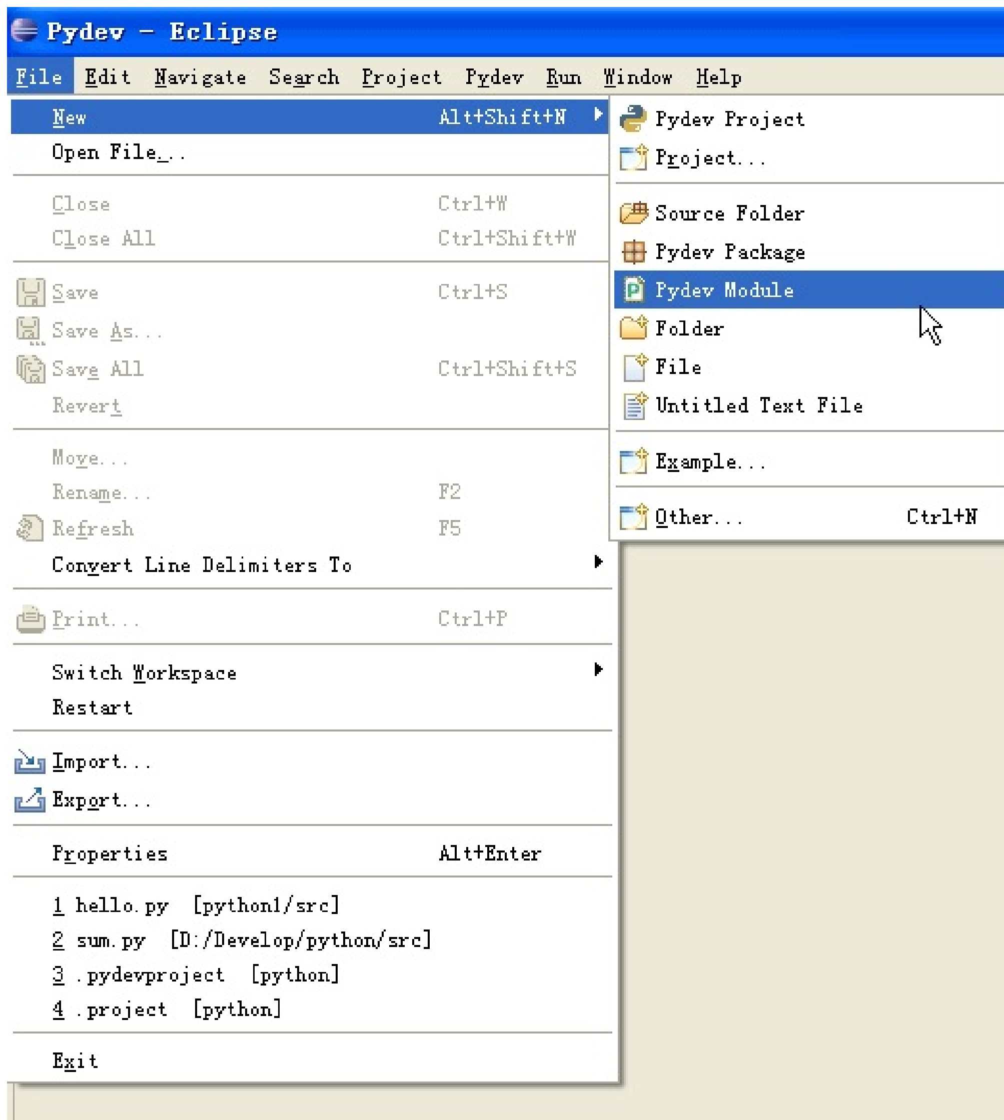


会弹出一个新窗口，填写Project Name，以及项目保存地址，然后点击next完成项目的创建。

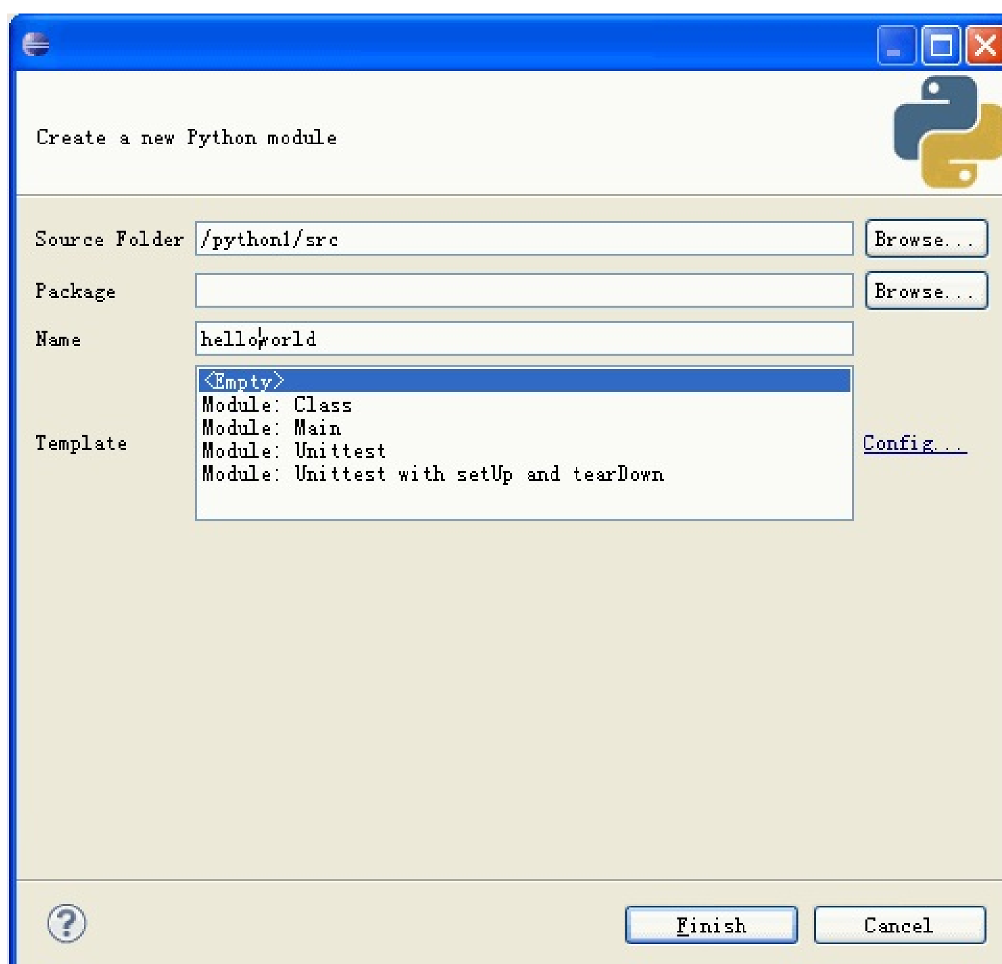


5、 创建新的Pydev Module

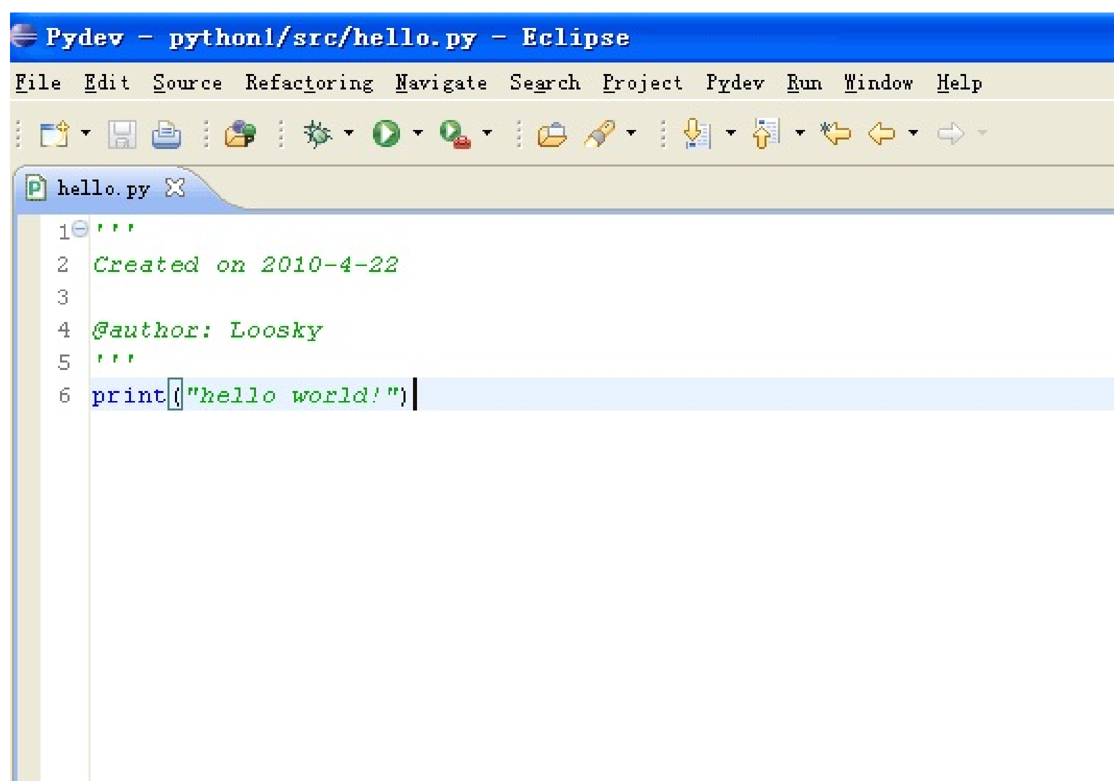
光有项目是无法执行的，接着必须创建新的Pydev Moudle，选择File -> New -> Pydev Module



在弹出的窗口中选择文件存放位置以及Moudle Name，注意Name不用加.py，它会自动帮助我们添加。然后点击Finish完成创建。

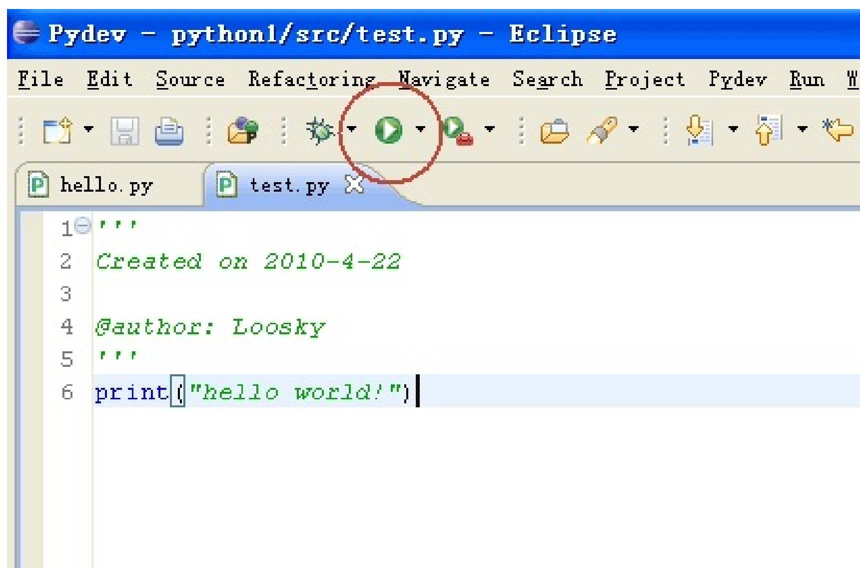


输入"hello world"的代码。

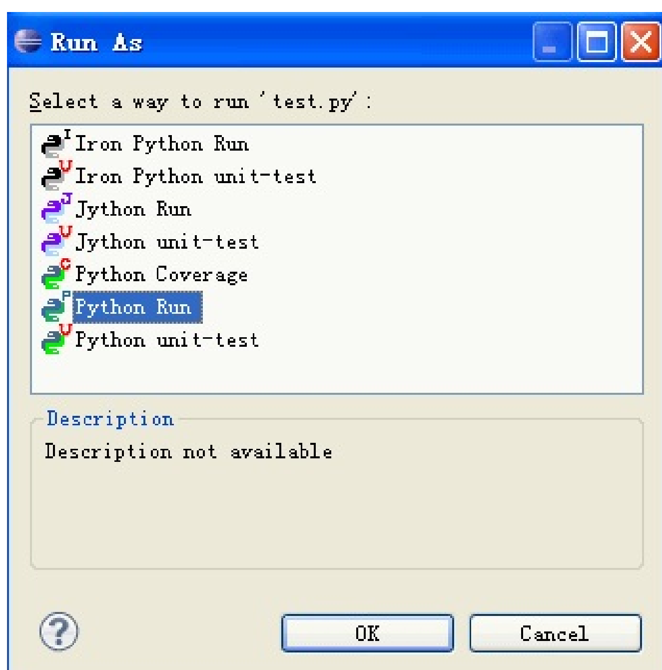


6、 执行程序

程序写完后，我们可以开始执行程序,在上方的工具栏上面找到执行的按钮。



之后会弹出一个让你选择执行方式的窗口，通常我们选择Python Run，开始执行程序。



更多 Python IDE

当然还有非常多很棒的 Python IDE，你可以自由的选择，更多 Python IDE 请参阅：http://wiki.python.org/moin/PythonEditors

Python JSON

本章节我们将为大家介绍如何使用 Python 语言来编码和解码 JSON 对象。

环境配置

在使用 Python 编码或解码 JSON 数据前，我们需要先安装 JSON 模块。本教程我们会下载 Demjson 并安装：

```
$tar xvfz demjson-1.6.tar.gz
$cd demjson-1.6
$python setup.py install
```

JSON 函数

函数	描述
encode	将 Python 对象编码成 JSON 字符串
decode	将已编码的 JSON 字符串解码为 Python 对象

encode

Python encode() 函数用于将 Python 对象编码成 JSON 字符串。

语法

```
demjson.encode(self, obj, nest_level=0)
```

实例

以下实例将数组编码为 JSON 格式数据：

```
#!/usr/bin/python
import demjson

data = [ { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4, 'e' : 5 } ]

json = demjson.encode(data)
print json
```

以上代码执行结果为：

```
[{"a":1,"b":2,"c":3,"d":4,"e":5}]
```

decode

Python 可以使用 demjson.decode() 函数解码 JSON 数据。该函数返回 Python 字段的数据类型。

语法

```
demjson.decode(self, txt)
```

实例

以下实例展示了Python 如何解码 JSON 对象：

```
#!/usr/bin/python
import demjson

json = '{"a":1,"b":2,"c":3,"d":4,"e":5}';

text = demjson.decode(json)
print  text
```

以上代码执行结果为：

```
{u'a': 1, u'c': 3, u'b': 2, u'e': 5, u'd': 4}
```

Python3 教程

Python3 基础语法

编码

默认情况下，Python 3源码文件以 UTF-8 编码，所有字符串都是 unicode 字符串。当然你也可以为源码文件指定不同的编码：

```
# -*- coding: cp-1252 -*-
```

标识符

- 第一个字符必须是字母表中字母或下划线'_'。
- 标识符的其他的部分有字母、数字和下划线组成。
- 标识符对大小写敏感。

在Python 3中，非-ASCII 标识符也是允许的了。

python保留字

保留字即关键字，我们不能把它们用作任何标识符名称。Python的标准库提供了一个keyword module，可以输出当前版本的所有关键字：

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
```

注释

Python中单行注释以#开头，多行注释用三个单引号（'''）或者三个双引号（"""）将注释括起来。

行与缩进

python最具特色的就是使用缩进来表示代码块。缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数。

数据类型

python中数有四种类型：整数、长整数、浮点数和复数。

- 整数， 如 1
- 长整数 是比较大的整数
- 浮点数 如 1.23、3E-2
- 复数 如 1 + 2j、 1.1 + 2.2j

字符串

- python中单引号和双引号使用完全相同。
- 使用三引号('''或""")可以指定一个多行字符串。
- 转义符 '\'
- 自然字符串， 通过在字符串前加r或R。 如 r"this is a line with \n" 则\n会显示，并不是换行。
- python允许处理unicode字符串，加前缀u或U， 如 u"this is an unicode string"。
- 字符串是不可变的。
- 按字面意义级联字符串，如"this " "is " "string"会被自动转换为this is string。

Python3 基本数据类型

Python中的变量不需要声明。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。

在Python中，变量就是变量，它没有类型，我们所说的"类型"是变量所指的内存中对象的类型。

Python 3中有六个标准的数据类型：

- Numbers（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Sets（集合）
- Dictionaries（字典）

Numbers（数字）

Python 3支持int、float、bool、complex（复数）。

数值类型的赋值和计算都是很直观的，就像大多数语言一样。内置的type()函数可以用来查询变量所指的对象类型。

```
>>> a, b, c, d = 20, 5.5, True, 4+3j
>>> print(type(a), type(b), type(c), type(d))
<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>
```

数值运算：

```
>>> 5 + 4  # 加法
9
>>> 4.3 - 2 # 减法
2.3
>>> 3 * 7  # 乘法
21
>>> 2 / 4  # 除法，得到一个浮点数
0.5
>>> 2 // 4 # 除法，得到一个整数
0
>>> 17 % 3 # 取余
2
>>> 2 ** 5 # 乘方
32
```

注意：

- 1、Python可以同时为多个变量赋值，如a, b = 1, 2。
- 2、一个变量可以通过赋值指向不同类型的对象。
- 3、数值的除法（/）总是返回一个浮点数，要获取整数使用//操作符。
- 4、在混合计算时，Pyhton会把整型转换成为浮点数。

String（字符串）

Python中的字符串str用单引号(' ')或双引号(" ")括起来，同时使用反斜杠()转义特殊字符。

```
>>> s = 'Yes,he doesn\'t'
>>> print(s, type(s), len(s))
Yes,he doesn't <class 'str'> 14
```

如果你不想让反斜杠发生转义，可以在字符串前面添加一个r，表示原始字符串：

```
>>> print('C:\some\name')
C:\some
ame
>>> print(r'C:\some\name')
C:\some\name
```

另外，反斜杠可以作为续行符，表示下一行是上一行的延续。还可以使用"""..."""或者'''...'''跨越多行。

字符串可以使用 + 运算符串连接在一起，或者用 * 运算符重复：

```
>>> print('str'+'ing', 'my'*3)
string mymymy
```

Python中的字符串有两种索引方式，第一种是从左往右，从0开始依次增加；第二种是从右往左，从-1开始依次减少。

注意，没有单独的字符类型，一个字符就是长度为1的字符串。

```
>>> word = 'Python'
>>> print(word[0], word[5])
P n
>>> print(word[-1], word[-6])
n P
```

还可以对字符串进行切片，获取一段子串。用冒号分隔两个索引，形式为变量[头下标:尾下标]。

截取的范围是前闭后开的，并且两个索引都可以省略：

```
>>> word = 'ilovepython'
>>> word[1:5]
'love'
>>> word[:]
'ilovepython'
>>> word[5:]
'python'
>>> word[-10:-6]
'love'
```

与C字符串不同的是，Python字符串不能被改变。向一个索引位置赋值，比如word[0] = 'm'会导致错误。

注意：

- 1、反斜杠可以用来转义，使用r可以让反斜杠不发生转义。
- 2、字符串可以用+运算符连接在一起，用*运算符重复。
- 3、Python中的字符串有两种索引方式，从左往右以0开始，从右往左以-1开始。
- 4、Python中的字符串不能改变。

List（列表）

List（列表） 是 Python 中使用最频繁的数据类型。

列表是写在方括号之间、用逗号分隔开的元素列表。列表中元素的类型可以不相同：

```
>>> a = ['him', 25, 100, 'her']
>>> print(a)
['him', 25, 100, 'her']
```

和字符串一样，列表同样可以被索引和切片，列表被切片后返回一个包含所需元素的新列表。详细的在这里就不赘述了。

列表还支持串联操作，使用+操作符：

```
>>> a = [1, 2, 3, 4, 5]
>>> a + [6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

与Python字符串不一样的是，列表中的元素是可以改变的：

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> a[0] = 9
>>> a[2:5] = [13, 14, 15]
>>> a
[9, 2, 13, 14, 15, 6]
>>> a[2:5] = []   # 删除
>>> a
[9, 2, 6]
```

List内置了有很多方法，例如append()、pop()等等，这在后面会讲到。

注意：

- 1、List写在方括号之间，元素用逗号隔开。
- 2、和字符串一样，list可以被索引和切片。
- 3、List可以使用+操作符进行拼接。
- 4、List中的元素是可以改变的。

Tuple（元组）

元组（tuple）与列表类似，不同之处在于元组的元素不能修改。元组写在小括号里，元素之间用逗号隔开。

元组中的元素类型也可以不相同：

```
>>> a = (1991, 2014, 'physics', 'math')
>>> print(a, type(a), len(a))
(1991, 2014, 'physics', 'math') <class 'tuple'> 4
```

元组与字符串类似，可以被索引且下标索引从0开始，也可以进行截取/切片（看上面，这里不再赘述）。

其实，可以把字符串看作一种特殊的元组。

```
>>> tup = (1, 2, 3, 4, 5, 6)
>>> print(tup[0], tup[1:5])
1 (2, 3, 4, 5)
>>> tup[0] = 11  # 修改元组元素的操作是非法的
```

虽然tuple的元素不可改变，但它可以包含可变的对象，比如list列表。

构造包含0个或1个元素的tuple是个特殊的问题，所以有一些额外的语法规则：

```
tup1 = () # 空元组
tup2 = (20,) # 一个元素，需要在元素后添加逗号
```

另外，元组也支持用+操作符：

```
>>> tup1, tup2 = (1, 2, 3), (4, 5, 6)
>>> print(tup1+tup2)
(1, 2, 3, 4, 5, 6)
```

string、list和tuple都属于sequence（序列）。

注意：

- 1、与字符串一样，元组的元素不能修改。
- 2、元组也可以被索引和切片，方法一样。
- 3、注意构造包含0或1个元素的元组的特殊语法规则。
- 4、元组也可以使用+操作符进行拼接。

Sets（集合）

集合（set）是一个无序不重复元素的集。

基本功能是进行成员关系测试和消除重复元素。

可以使用大括号 或者 set()函数创建set集合，注意：创建一个空集合必须用 set() 而不是 { }，因为{ }是用来创建一个空字典。

```
>>> student = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'}
>>> print(student)   # 重复的元素被自动去掉
{'Jim', 'Jack', 'Mary', 'Tom', 'Rose'}
>>> 'Rose' in student  # membership testing（成员测试）
True
>>> # set可以进行集合运算
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'b', 'c', 'd', 'r'}
>>> a - b     # a和b的差集
{'b', 'd', 'r'}
>>> a | b     # a和b的并集
{'l', 'm', 'a', 'b', 'c', 'd', 'z', 'r'}
>>> a & b     # a和b的交集
{'a', 'c'}
>>> a ^ b     # a和b中不同时存在的元素
{'l', 'm', 'b', 'd', 'z', 'r'}
```

Dictionaries（字典）

字典（dictionary）是Python中另一个非常有用的内置数据类型。

字典是一种映射类型（mapping type），它是一个无序的键 : 值对集合。

关键字必须使用不可变类型，也就是说list和包含可变类型的tuple不能做关键字。

在同一个字典中，关键字还必须互不相同。

```
>>> dic = {}  # 创建空字典
>>> tel = {'Jack':1557, 'Tom':1320, 'Rose':1886}
>>> tel
{'Tom': 1320, 'Jack': 1557, 'Rose': 1886}
>>> tel['Jack']   # 主要的操作：通过key查询
1557
>>> del tel['Rose']  # 删除一个键值对
>>> tel['Mary'] = 4127  # 添加一个键值对
>>> tel
{'Tom': 1320, 'Jack': 1557, 'Mary': 4127}
>>> list(tel.keys())  # 返回所有key组成的list
['Tom', 'Jack', 'Mary']
>>> sorted(tel.keys()) # 按key排序
['Jack', 'Mary', 'Tom']
>>> 'Tom' in tel       # 成员测试
True
>>> 'Mary' not in tel  # 成员测试
False
```

构造函数 dict() 直接从键值对sequence中构建字典，当然也可以进行推导，如下：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'jack': 4098, 'sape': 4139, 'guido': 4127}

>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}

>>> dict(sape=4139, guido=4127, jack=4098)
{'jack': 4098, 'sape': 4139, 'guido': 4127}
```

另外，字典类型也有一些内置的函数，例如clear()、keys()、values()等。

注意：

- 1、字典是一种映射类型，它的元素是键值对。
- 2、字典的关键字必须为不可变类型，且不能重复。
- 3、创建空字典使用{ }。

Python解释器

Linux/Unix的系统上，Python解释器通常被安装在 /usr/local/bin/python3.4 这样的有效路径（目录）里。

我们可以将路径 /usr/local/bin 添加到您的Linux/Unix操作系统的环境变量中，这样您就可以通过 shell 终端输入下面的命令来启动 Python 。

```
python3.4
```

在Window系统下你可以通过以下命令来设置Python的环境变量，假设你的Python安装在 C:\Python34 下:

```
set path=%path%;C:\python34
```

交互式编程

我们可以在命令提示符中输入"Python"命令来启动Python解释器：

```
python
```

执行以上命令后，出现如下窗口信息：

```
$ python3.4
Python 3.4 (default, Mar 16 2014, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more informat:
>>>
```

在 python 提示符中输入以下语句，然后按回车键查看运行效果：

```
print ("Hello, Python!");
```

以上命令执行结果如下：

```
Hello, Python!
```

当键入一个多行结构时，续行是必须的。我们可以看下如下 if 语句：

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

脚本式编程

将如下代码拷贝至hello.py文件中：

```
print ("Hello, Python!");
```

通过以下命令执行该脚本：

```
python hello.py
```

输出结果为：

```
Hello, Python!
```

在Linux/Unix系统中，你可以在脚本顶部添加以下命令让Python脚本可以像SHELL脚本一样可直接执行：

```
#! /usr/bin/env python3.4
```

然后修改脚本权限，使其有执行权限，命令如下：

```
$ chmod +x hello.py
```

执行以下命令：

```
./hello.py
```

输出结果为：

```
Hello, Python!
```

有关Python基础语法部分请参阅：Python基础语法

Python 注释

确保对模块, 函数, 方法和行内注释使用正确的风格

Python中的注释有单行注释和多行注释：

Python中单行注释以#开头，例如：：

```
# 这是一个注释
print("Hello, World!")
```

多行注释用三个单引号（'''）或者三个双引号（"""）将注释括起来，例如:

1、单引号（'''）

```
#!/usr/bin/python3
'''
这是多行注释，用三个单引号
这是多行注释，用三个单引号
这是多行注释，用三个单引号
'''
print("Hello, World!")
```

2、双引号（'''）

```
#!/usr/bin/python3
"""
这是多行注释，用三个单引号
这是多行注释，用三个单引号
这是多行注释，用三个单引号
"""
print("Hello, World!")
```

Python 数字运算

Python 解释器可以作为一个简单的计算器：您可以在解释器里输入一个表达式，它将输出表达式的值。

表达式的语法很直白： +, -, * 和/ 和在许多其它语言（如Pascal或C）里一样；括号可以用来为运算分组。例如：

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # 总是返回一个浮点数
1.6
```

注意：在不同的机器上浮点运算的结果可能会不一样。之后我们会介绍有关控制浮点运算输出结果的内容。

在整数除法中，除法（/）总是返回一个浮点数，如果只想得到整数的结果，丢弃可能的分数部分，可以使用运算符 // ：

```
>>> 17 / 3  # 整数除法返回浮点型
5.666666666666667
>>>
>>> 17 // 3  # 整数除法返回向下取整后的结果
5
>>> 17 % 3  # %操作符返回除法的余数
2
>>> 5 * 3 + 2
17
```

等号（'='）用于给变量赋值。赋值之后，除了下一个提示符，解释器不会显示任何结果。

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Python 可以使用**操作来进行幂运算：

```
>>> 5 ** 2  # 5 的平方
25
>>> 2 ** 7  # 2的7次方
128
```

变量在使用前必须先"定义"（即赋予变量一个值），否则会出现错误：

```
>>> # 尝试访问一个未定义的变量
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

浮点数得到完全的支持；不同类型的数混合运算时会将整数转换为浮点数：

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

在交互模式中，最后被输出的表达式结果被赋值给变量 _ 。这能使您在把Python作为一个桌面计算器使用时使后续计算更方便，例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

此处， _ 变量应被用户视为只读变量。不要显式地给它赋值——这样您将会创建一个具有相同名称的独立的本地变量，并且屏蔽了这个内置变量的功能。

Python 字符串

除了数字，Python也能操作字符串。字符串有几种表达方式，可以使用单引号或双引号括起来：

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Python中使用反斜杠转义引号和其它特殊字符来准确地表示。

如果字符串包含有单引号但不含双引号，则字符串会用双引号括起来，否则用单引号括起来。对于这样的输入字符串，print() 函数会产生更易读的输出。

跨行的字面字符串可用以下几种方法表示。使用续行符，即在每行最后一个字符后使用反斜线来说明下一行是上一行逻辑上的延续：

以下使用 \n 来添加新行：

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'First line.\nSecond line.'  # \n 意味着新行
>>> s  # 不使用 print(), \n 包含在输出中
'First line.\nSecond line.'
>>> print(s)  # 使用 print(), \n 输出一个新行
First line.
Second line.
```

以下使用 反斜线（\） 来续行：

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
 significant."

print(hello)
```

注意，其中的换行符仍然要使用 \n 表示——反斜杠后的换行符被丢弃了。以上例子将如下输出：

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significar
```

或者，字符串可以被 """ （三个双引号）或者 ''' （三个单引号）括起来。使用三引号时，换行符不需要转义，它们会包含在字符串中。以下的例子使用了一个转义符，避免在最开始产生一个不需要的空行。

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

其输出如下：

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

如果我们使用"原始"字符串，那么 \n 不会被转换成换行，行末的的反斜杠，以及源码中的换行符，都将作为数据包含在字符串内。例如：

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print(hello)
```

将会输出：

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

字符串可以使用 + 运算符串连接在一起，或者用 * 运算符重复：

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

两个紧邻的字面字符串将自动被串连；上例的第一行也可以写成 word = 'Help' 'A' ；这样的操作只在两个字面值间有效，不能随意用于字符串表达式中：

```
>>> 'str' 'ing'                   #  <-  这样操作正确
'string'
>>> 'str'.strip() + 'ing'   #  <-  这样操作正确
'string'
>>> 'str'.strip() 'ing'     #  <-  这样操作错误
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                      ^
SyntaxError: invalid syntax
```

字符串可以被索引；就像 C 语言一样，字符串的第一个字符的索引为 0。没有单独的字符类型；一个字符就是长度为一的字符串。就像Icon编程语言一样，子字符串可以使用分切符来指定：用冒号分隔的两个索引。

```
>>> word[4]
'A'
>>> word[0:2]
'Hl'
>>> word[2:4]
'ep'
```

默认的分切索引很有用：默认的第一个索引为零，第二个索引默认为字符串可以被分切的长度。

```
>>> word[:2]    # 前两个字符
'He'
>>> word[2:]    # 除了前两个字符之外，其后的所有字符
'lpA'
```

不同于C字符串的是，Python字符串不能被改变。向一个索引位置赋值会导致错误：

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'str' object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'str' object does not support slice assignment
```

然而，用组合内容的方法来创建新的字符串是简单高效的：

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
在分切操作字符串时，有一个很有用的规律： s[:i] + s[i:] 等于 s.

>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

对于有偏差的分切索引的处理方式也很优雅：一个过大的索引将被字符串的大小取代，上限值小于下限值将返回一个空字符串。

```
>>> word[1:100]
'elpA'
>>> word[10:]

>>> word[2:1]
```

在索引中可以使用负数，这将会从右往左计数。例如：

```
>>> word[-1]     # 最后一个字符
'A'
>>> word[-2]     # 倒数第二个字符
'p'
>>> word[-2:]    # 最后两个字符
'pA'
>>> word[:-2]    # 除了最后两个字符之外，其前面的所有字符
'Hel'
但要注意， -0 和 0 完全一样，所以 -0 不会从右开始计数！

>>> word[-0]     # (既然 -0 等于 0)
'H'
```

超出范围的负数索引会被截去多余部分，但不要尝试在一个单元素索引（非分切索引）里使用：

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # 错误
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

有一个方法可以让您记住分切索引的工作方式，想像索引是指向字符之间，第一个字符左边的数字是 0。接着，有n个字符的字符串最后一个字符的右边是索引n，例如：

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

第一行的数字 0...5 给出了字符串中索引的位置；第二行给出了相应的负数索引。分切部分从 i 到 j 分别由在边缘被标记为 i 和 j 的全部字符组成。

对于非负数分切部分，如果索引都在有效范围内，分切部分的长度就是索引的差值。例如， word[1:3] 的长度是2。

内置的函数 len() 用于返回一个字符串的长度：

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Python 列表

Python囊括了大量的复合数据类型，用于组织其它数值。最有用的是列表，即写在方括号之间、用逗号分隔开的数值列表。列表内的项目不必全是相同的类型。

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

像字符串一样，列表可以被索引和切片：

```
<pre>
>>> squares[0]  # 索引返回的指定项
1
>>> squares[-1]
25
>>> squares[-3:]  # 切割列表并返回新的列表
[9, 16, 25]
```

所有的分切操作返回一个包含有所需元素的新列表。如下例中，分切将返回列表 squares 的一个拷贝：

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

列表还支持拼接操作：

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Python 字符串是固定的，列表可以改变其中的元素：

```
>>> cubes = [1, 8, 27, 65, 125]
>>> 4 ** 3
64
>>> cubes[3] = 64  # 修改列表值
>>> cubes
[1, 8, 27, 64, 125]
```

您也可以通过使用append()方法在列表的末尾添加新项：

```
>>> cubes.append(216)  # cube列表中添加新值
>>> cubes.append(7 ** 3)  #  cube列表中添加第七个值
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

你也可以修改指定区间的列表值：

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # 替换一些值
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # 移除值
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # 清楚列表
>>> letters[:] = []
>>> letters
[]
```

内置函数 len() 用于统计列表：

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

也可以使用嵌套列表（在列表里创建其它列表），例如：

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Python 编程第一步

现在，我们能使用Python完成比 2+2 更复杂的工作。在下例里，我们能写出一个初步的斐波纳契数列如下：

```
>>> # Fibonacci series: 斐波纳契数列
... # 两个元素的总和确定了下一个数
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

这个例子介绍了几个新特征。

- 第一行包含了一个复合赋值：变量 a 和 b 同时得到新值 0 和 1。最后一行再次使用了同样的方法，可以看到，右边的表达式会在赋值变动之前执行。右边表达式的执行顺序是从左往右的。

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

关键字end可以被用于防止输出新的一行，或者在输出的末尾添加不同的字符：

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=',')
...     a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Python 条件控制

if 语句

Python中if语句的一般形式如下所示：

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

如果 "condition_1" 为 True 将执行 "statement_block_1" 块语句，如果 "condition_1" 为False，将判断 "condition_2"，如果"condition_2" 为 True 将执行 "statement_block_2" 块语句，如果 "condition_2" 为False，将执行"statement_block_3"块语句。

Python中用elif代替了else if，所以if语句的关键字为：if – elif – else。

注意：

- 1、每个条件后面要使用冒号（:），表示接下来是满足条件后要执行的语句块。
- 2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。
- 3、在Python中没有switch – case语句。

实例

以下实例演示了狗的年龄计算判断：

```
age = int(input("Age of the dog: "))
print()
if age < 0:
    print("This can hardly be true!")
elif age == 1:
    print("about 14 human years")
elif age == 2:
    print("about 22 human years")
elif age > 2:
    human = 22 + (age -2)*5
    print("Human years: ", human)

###
input('press Return>')
```

将以上脚本保存在dog.py文件中，并执行该脚本：

```
python dog.py
Age of the dog: 1

about 14 human years
```

以下为if中常用的操作运算符:

操作符	描述
`<`	小于
`<=`	小于或等于
`>`	大于
`>=`	大于或等于
`==`	等于，比较对象是否相等
`!=`	不等于

实例

```
# 程序演示了 == 操作符
# 使用数字
print(5 == 6)
# 使用变量
x = 5
y = 8
print(x == y)
```

以上实例输出结果：

```
False
False
```

high_low.py文件：

```
#!/usr/bin/python3
# 该实例演示了数字猜谜游戏
number = 7
guess = -1
print("Guess the number!")
while guess != number:
    guess = int(input("Is it... "))

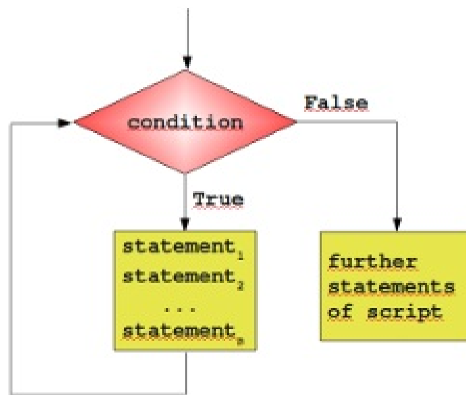
    if guess == number:
        print("Hooray! You guessed it right!")
    elif guess < number:
        print("It's bigger...")
    elif guess > number:
        print("It's not so big.")
```

Python 循环

本章节将为大家介绍Python循环语句的使用。

Python中的循环语句有 for 和 while。

Python循环语句的控制结构图如下所示：



while 循环

Python中while语句的一般形式：

```
while 判断条件：
    statements
```

同样需要注意冒号和缩进。另外，在Python中没有do..while循环。

以下实例使用了 while 来计算 1 到 100 的总和：

```
#!/usr/bin/env python3

n = 100

sum = 0
counter = 1
while counter <= n:
    sum = sum + counter
    counter += 1

print("Sum of 1 until %d: %d" % (n,sum))
```

执行结果如下：

```
Sum of 1 until 100: 5050
```

for语句

Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

for循环的一般格式如下：

```
for <variable> in <sequence>:
    <statements>
else:
    <statements>
```

Python loop循环实例：

```
>>> languages = ["C", "C++", "Perl", "Python"]
>>> for x in languages:
...     print x
...
C
C++
Perl
Python
>>>
```

以下实例for实例中使用了 break语句，break语句用于跳出当前循环体：

```
#!/usr/bin/env python3
edibles = ["ham", "spam","eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

执行脚本后，在循环到 "spam"时会跳出循环体：

```
Great, delicious ham
No more spam please!
Finally, I finished stuffing myself
```

range()函数

如果你需要遍历数字序列，可以使用内置range()函数。它会生成数列，例如:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

你也可以使用range指定区间的值：

```
>>> for i in range(5,9) :
    print(i)

5
6
7
8
>>>
```

也可以使range以指定数字开始并指定不同的增量(甚至可以是负数;有时这也叫做'步长'):

```
>>> for i in range(0, 10, 3) :
    print(i)

0
3
6
9
>>>
```

负数：

```
>>> for i in range(-10, -100, -30) :
    print(i)

-10
-40
-70
>>>
```

您可以结合range()和len()函数以遍历一个序列的索引,如下所示:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

还可以使用range()函数来创建一个列表：

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>>
```

break和continue语句及循环中的else子句

break语句可以跳出for和while的循环体。如果你从for或while循环中终止，任何对应的循环else块将不执行。

continue语句被用来告诉Python跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

循环语句可以有else子句;它在穷尽列表(以for循环)或条件变为假(以while循环)循环终止时被执行,但循环被break终止时不执行.如下查寻质数的循环例子:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # 循环中没有找到元素
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

pass语句

pass语句什么都不做。它只在语法上需要一条语句但程序不需要任何操作时使用.例如:

```
>>> while True:
...     pass  # 等待键盘中断 (Ctrl+C)
```

最小的类:

```
>>> class MyEmptyClass:
...     pass
```

Python 函数

本章节我们将为大家介绍Python中函数的应用。

该章节可参阅Python 函数应用详解。

Python 定义函数使用 def 关键字，一般格式如下：

```
def  函数名（参数列表）：
    函数体
```

让我们使用函数来输出"Hello World！"：

```
>>> def hello() :
    print("Hello World!")

>>> hello()
Hello World!
>>>
```

更复杂点的应用，函数中带上参数变量:

```
def area(width, height):
    return width * height

def print_welcome(name):
    print("Welcome", name)

print_welcome("Fred")
w = 4
h = 5
print("width =", w, " height =", h, " area =", area(w, h))
```

以上实例输出结果：

```
Welcome Fred
width = 4  height = 5  area = 20
```

函数变量作用域

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

通过以下实例，你可以清楚了解Python函数变量的作用域：

```
#!/usr/bin/env python3
a = 4  # 全局变量

def print_func1():
    a = 17 # 局部变量
    print("in print_func a = ", a)
def print_func2():
    print("in print_func a = ", a)
print_func1()
print_func2()
print("a = ", a)
```

以上实例运行结果如下：

```
in print_func a =  17
in print_func a =  4
a =  4
```

关键字参数

函数也可以使用 kwarg=value 的关键字参数形式被调用.例如,以下函数:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegiar
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

可以以下几种方式被调用:

```
parrot(1000)                                          # 1 positiona
parrot(voltage=1000)                                  # 1 keyword a
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword a
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword a
parrot('a million', 'bereft of life', 'jump')         # 3 positiona
parrot('a thousand', state='pushing up the daisies')  # 1 positiona
```

以下为错误调用方法：

```
parrot()                         # required argument missing
parrot(voltage=5.0, 'dead')   # non-keyword argument after a keyword
parrot(110, voltage=220)      # duplicate value for the same argumen
parrot(actor='John Cleese')   # unknown keyword argument
```

返回值

Python的函数的返回值使用return语句，可以将函数作为一个值赋值给指定变量：

```
def return_sum(x,y):
    c = x + y
    return c

res = return_sum(4,5)
print(res)
```

你也可以让函数返回空值：

```
def empty_return(x,y):
    c = x + y
    return

res = empty_return(4,5)
print(res)
```

可变参数列表

最后,一个最不常用的选择是可以让函数调用可变个数的参数.这些参数被包装进一个元组(查看元组和序列).在这些可变个数的参数之前,可以有零到多个普通的参数:

```
def arithmetic_mean(*args):
    sum = 0
    for x in args:
        sum += x
    return sum

print(arithmetic_mean(45,32,89,78))
print(arithmetic_mean(8989.8,78787.78,3453,78778.73))
print(arithmetic_mean(45,32))
print(arithmetic_mean(45))
print(arithmetic_mean())
```

以上实例输出结果为：

```
244
170009.31
77
45
0
```

更详细教程请参阅参阅Python 函数应用详解。

Python 数据结构

本章节我们主要结合前面所学的知识点来介绍Python数据结构。

列表

Python中列表是可变的，这是它区别于字符串和元组的最重要的特点，一句话概括即：列表可以修改，而字符串和元组不能。

以下是 Python 中列表的方法：

方法	描述
list.append(x)	把一个元素添加到列表的结尾，相当于 a[len(a):] = [x]。
list.extend(L)	通过添加指定列表的所有元素来扩充列表，相当于 a[len(a):] = L。
list.insert(i, x)	在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如 a.insert(0, x) 会插入到整个列表之前，而 a.insert(len(a), x) 相当于 a.append(x) 。
list.remove(x)	删除列表中值为 x 的第一个元素。如果没有这样的元素，就会返回一个错误。
list.pop([i])	从列表的指定位置删除元素，并将其返回。如果没有指定索引，a.pop()返回最后一个元素。元素随即从列表中被删除。（方法中 i 两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在 Python 库参考手册中遇到这样的标记。）
list.clear()	移除列表中的所有项，等于del a[:]。
list.index(x)	返回列表中第一个值为 x 的元素的索引。如果没有匹配的元素就会返回一个错误。
list.count(x)	返回 x 在列表中出现的次数。
list.sort()	对列表中的元素进行排序。
list.reverse()	倒排列表中的元素。
list.copy()	返回列表的浅复制，等于a[:]。

下面示例演示了列表的大部分方法：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

注意：类似 insert, remove 或 sort 等修改列表的方法没有返回值。

将列表当做堆栈使用

列表方法使得列表可以很方便的作为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。用 append() 方法可以把一个元素添加到堆栈顶。用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放出来。例如：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

将列表当作队列使用

也可以把列表当做队列用，只是在队列里第一加入的元素，第一个取出来；但是拿列表用作这样的目的效率不高。在列表的最后添加或者弹出元素速度快，然而在列表里插入或者从头部弹出速度却不快（因为所有其他的元素都得一个一个地移动）。

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leave
'Eric'
>>> queue.popleft()                 # The second to arrive now leav
'John'
>>> queue                           # Remaining queue in order of a
deque(['Michael', 'Terry', 'Graham'])
```

列表推导式

列表推导式提供了从序列创建列表的简单途径。通常应用程序将一些操作应用于某个序列的每个元素，用其获得的结果作为生成新列表的元素，或者根据确定的判定条件创建子序列。

每个列表推导式都在 for 之后跟一个表达式，然后有零到多个 for 或 if 子句。返回结果是一个根据表达从其后的 for 和 if 上下文环境中生成出来的列表。如果希望表达式推导出一个元组，就必须使用括号。

这里我们将列表中每个数值乘三，获得一个新的列表：

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

现在我们玩一点小花样：

```
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

这里我们对序列里每一个元素逐个调用某方法：

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

我们可以用 if 子句作为过滤器：

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

以下是一些关于循环和其它技巧的演示：

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

列表推导式可以使用复杂表达式或嵌套函数：

```
>>> [str(round(355/113, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

嵌套列表解析

Python的列表还可以嵌套。

以下实例展示了3X4的矩阵列表：

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

以下实例将3X4的矩阵列表转换为4X3列表：

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

以下实例也可以使用以下方法来实现：

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

另外一种实现方法：

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

del 语句

使用 del 语句可以从一个列表中依索引而不是值来删除一个元素。这与使用 pop() 返回一个值不同。可以用 del 语句从列表中删除一个切割，或清空整个列表（我们以前介绍的方法是给该切割赋一个空列表）。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

也可以用 del 删除实体变量：

```
>>> del a
```

元组和序列

元组由若干逗号分隔的值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可能有或没有括号， 不过括号通常是必须的（如果元组是更大的表达式的一部分）。

集合

集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。

可以用大括号({})创建集合。注意：如果要创建一个空集合，你必须用 set() 而不是 {} ；后者创建一个空的字典，下一节我们会介绍这个数据结构。

以下是一个简单的演示：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana
>>> print(basket)                      # show that duplicates have
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not
{'r', 'd', 'b', 'm', 'z', 'l'}>>> basket = {'apple', 'orange', 'app
>>> print(basket)                      # show that duplicates have
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not
{'r', 'd', 'b', 'm', 'z', 'l'}
```

集合也支持推导式：

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

字典

另一个非常有用的 Python 内建数据类型是字典。

序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。

理解字典的最佳方式是把它看做无序的键=>值对集合。在同一个字典之内，关键字必须是互不相同。

一对大括号创建一个空的字典：{}。

这是一个字典运用的简单例子：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

构造函数 dict() 直接从键值对元组列表中构建字典。如果有固定的模式，列表推导式指定特定的键值对：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

此外，字典推导可以用来创建任意键和值的表达式词典：

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

如果关键字只是简单的字符串，使用关键字参数指定键值对有时候更方便：

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

遍历技巧

在字典中遍历时，关键字和对应的值可以使用 items() 方法同时解读出来：

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

在序列中遍历时，索引位置和对应值可以使用 enumerate() 函数同时得到：

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

同时遍历两个或更多的序列，可以使用 zip() 组合：

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

要反向遍历一个序列，首先指定这个序列，然后调用 reversesd() 函数：

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

要按顺序遍历一个序列，使用 sorted() 函数返回一个已排序的序列，并不修改原值：

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

其他参阅文档(Python2.x)

- Python 列表
- Python 元组
- Python 字典

Python 模块

在前面的几个章节中我们脚本上是用python解释器来编程，如果你从Python解释器退出再进入，那么你定义的所有的方法和变量就都消失了。

为此 Python 提供了一个办法，把这些定义存放在文件中，为一些脚本或者交互式的解释器实例使用，这个文件被称为模块。

模块是一个包含所有你定义的函数和变量的文件，其后缀名是.py。模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用python标准库的方法。下面是一个使用python标准库中模块的例子。

```
#!/usr/bin/python3
# Filename: using_sys.py

import sys

print('命令行参数如下:')
for i in sys.argv:
    print(i)

print('/n/nThe PYTHONPATH is', sys.path, '/n')
```

执行结果如下所示：

```
E:\python33\src>python using_sys.py 参数1 参数2
命令行参数如下:
using_sys.py
参数1
参数2
/n/nThe PYTHONPATH is ['E:\\python33\\src', 'C:\\Windows\\system32\
', 'E:\\python33\\DLLs', 'E:\\python33\\lib', 'E:\\python33', 'E:\\
\\site-packages'] /n
```

- 1、import sys引入python标准库中的sys.py模块；这是引入某一模块的方法。
- 2、sys.argv是一个包含命令行参数的列表。
- 3、sys.path包含了一个Python解释器自动查找所需模块的路径的列表。

当我们使用import语句的时候，Python解释器是怎样找到对应的文件的呢？

这就涉及到Python的搜索路径，搜索路径是由一系列目录名组成的，Python解释器就依次从这些目录中去寻找锁引入的模块。

这看起来很像环境变量，事实上，也可以通过定义环境变量的方式来确定搜索路径。

搜索路径是在Python编译或安装的时候确定的，安装新的库应该也会修改。搜索路径被存储在sys模块中的path变量，做一个简单的实验，在交互式解释器中，输入以下代码：

```
import sys
sys.path
```

输出结果：

```
>>> sys.path
['', 'E:\\python33\\Lib\\idlelib', 'C:\\Windows\\system32\\python33
```

sys.path输出是一个列表，其中第一项是空串''，代表当前目录（若是从一个脚本中打印出来的话，可以更清楚地看出是哪个目录），亦即我们执行python解释器的目录（对于脚本的话就是运行的脚本所在的目录）。

因此若像我一样在当前目录下存在与要引入模块同名的文件，就会把要引入的模块屏蔽掉。

了解了搜索路径的概念，就可以在脚本中修改sys.path来引入一些不在搜索路径中的模块。

现在，在解释器的当前目录或者sys.path中的一个目录里面来创建一个fibo.py的文件，代码如下：

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

然后进入Python解释器，使用下面的命令导入这个模块：

```
>>> import fibo
```

这样做并没有把直接定义在fibo中的函数名称写入到当前符号表里，只是把模块fibo的名字写到了那里。

可以使用模块名称来访问函数：

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

如果你打算经常使用一个函数，你可以把它赋给一个本地的名称：

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

深入模块

模块除了方法定义，还可以包括可执行的代码。这些代码一般用来初始化这个模块。这些代码只有在第一次被导入时才会被执行。

每个模块有各自独立的符号表，在模块内部为所有的函数当作全局符号表来使用。

所以，模块的作者可以放心大胆的在模块内部使用这些全局变量，而不用担心把其他用户的全局变量搞花。

从另一个方面，当你确实知道你在做什么的话，你也可以通过 modname.itemname 这样的表示法来访问模块内的函数。

模块是可以导入其他模块的。在一个模块（或者脚本，或者其他地方）的最前面使用 import 来导入一个模块，当然这只是一个惯例，而不是强制的。被导入的模块的名称将被放入当前操作的模块的符号表中。

还有一种导入的方法，可以使用 import 直接把模块内（函数，变量的）名称导入到当前操作模块。比如:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这种导入的方法不会把被导入的模块的名称放在当前的字符表中（所以在这个例子里面，fibo 这个名称是没有定义的）。

这还有一种方法，可以一次性的把模块中的所有（函数，变量）名称都导入到当前模块的字符表:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这将把所有的名字都导入进来，但是那些由单一下划线（_）开头的名字不在此例。大多数情况， Python程序员不使用这种方法，因为引入的其它来源的命名，很可能覆盖了已有的定义。

name属性

一个模块被另一个程序第一次引入时，其主程序将运行。如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用**name**属性来使该程序块仅在该模块自身运行时执行。

```
#!/usr/bin/python3
# Filename: using_name.py

if __name__ == '__main__':
    print('程序自身在运行')
else:
    print('我来自另一模块')
```

运行输出如下：

```
$ python using_name.py
```

程序自身在运行

```
$ python
>>> import using_name
我来自另一模块
>>>
```

说明： 每个模块都有一个**name**属性，当其值是**'main'**时，表明该模块自身在运行，否则是被引入。

dir() 函数

```
内置的函数 dir() 可以找到模块内定义的所有名称。以一个字符串列表的形式返回:
</p>
<pre>
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__r
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_get
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'ar
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteor
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursion
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionl
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

如果没有给定参数，那么 dir() 函数会罗列出当前定义的所有名称:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir() # 得到一个当前模块中定义的属性列表
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
>>> a = 5 # 建立一个新的变量 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'sys']
>>>
>>> del a # 删除变量名a
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'sys']
>>>
```

标准模块

Python 本身带着一些标准的模块库，在 Python 库参考文档中将会介绍到（就是后面的"库参考文档"）。

有些模块直接被构建在解析器里，这些虽然不是一些语言内置的功能，但是他却能很高效的使用，甚至是系统级调用也没问题。

这些组件会根据不同的操作系统进行不同形式的配置，比如 winreg 这个模块就只会提供给 Windows 系统。

应该注意到这有一个特别的模块 sys ，它内置在每一个 Python 解析器中。变量 sys.ps1 和 sys.ps2 定义了主提示符和副提示符所对应的字符串:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

包

包是一种管理 Python 模块命名空间的形式，采用"点模块名称"。

比如一个模块的名称是 A.B， 那么他表示一个包 A中的子模块 B 。

就好像使用模块的时候，你不用担心不同模块之间的全局变量相互影响一样，采用点模块名称这种形式也不用担心不同库之间的模块重名的情况。

这样不同的作者都可以提供 NumPy 模块，或者是 Python 图形库。

不妨假设你想设计一套统一处理声音文件和数据的模块（或者称之为一个"包"）。

现存很多种不同的音频文件格式（基本上都是通过后缀名区分的，例如： .wav，:file:.aiff，:file:.au，），所以你需要有一组不断增加的模块，用来在不同的格式之间转换。

并且针对这些音频数据，还有很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所你还需要一组怎么也写不完的模块来处理这些操作。

这里给出了一种可能的包结构（在分层的文件系统中）:

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversi
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

在导入一个包的时候，Python 会根据 sys.path 中的目录来寻找这个包中包含的子目录。

目录只有包含一个叫做 **init**.py 的文件才会被认作是一个包，主要是为了避免一些滥俗的名字（比如叫做 string）不小心的影响搜索路径中的有效模块。

最简单的情况，放一个空的 :file:**init**.py就可以了。当然这个文件中也可以包含一些初始化代码或者为（将在后面介绍的） **all**变量赋值。

用户可以每次只导入一个包里面的特定模块，比如:

```
import sound.effects.echo
```

这将会导入子模块:mod:song.effects.echo。 他必须使用全名去访问:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

还有一种导入子模块的方法是:

```
from sound.effects import echo
```

这同样会导入子模块:mod:echo，并且他不需要那些冗长的前缀，所以他可以这样使用:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

还有一种变化就是直接导入一个函数或者变量:

```
from sound.effects.echo import echofilter
```

同样的，这种方法会导入子模块:mod:echo，并且可以直接使用他的:func:echofilter函数:

echofilter(input, output, delay=0.7, atten=4)

注意当使用from package import item这种形式的时候，对应的item既可以是包里面的子模块（子包），或者包里面定义的其他名称，比如函数，类或者变量。

import语法会首先把item当作一个包定义的名称，如果没找到，再试图按照一个模块去导入。如果还没找到，恭喜，一个:exc:ImportError 异常被抛出了。

反之，如果使用形如import item.subitem.subsubitem这种导入形式，除了最后一项，都必须是包，而最后一项则可以是模块或者是包，但是不可以是类，函数或者变量的名字。

从一个包中导入*

设想一下，如果我们使用 from sound.effects import *会发生什么？

Python 会进入文件系统，找到这个包里面所有的子模块，一个一个的把它们都导入进来。

但是很不幸，这个方法在 Windows平台上工作的就不是非常好，因为Windows是一个大小写不区分的系统。

在这类平台上，没有人敢担保一个叫做 ECHO.py 的文件导入为模块:mod:echo还是:mod:Echo甚至:mod:ECHO。

（例如，Windows 95就很讨厌的把每一个文件的首字母大写显示）而且 DOS 的 8+3 命名规则对长模块名称的处理会把问题搞得更纠结。

为了解决这个问题，只能烦劳包作者提供一个精确的包的索引了。

导入语句遵循如下规则：如果包定义文件 **init**.py 存在一个叫做 **all** 的列表变量，那么在使用 from package import * 的时候就把这个列表中的所有名字作为包内容导入。

作为包的作者，可别忘了在更新包之后保证 **all** 也更新了啊。你说我就不这么做，我就不使用导入*这种用法，好吧，没问题，谁让你是老板呢。这里有一个例子，在:file:sounds/effects/**init**.py中包含如下代码:

```
__all__ = ["echo", "surround", "reverse"]
```

这表示当你使用from sound.effects import *这种用法时，你只会导入包里面这三个子模块。

如果**all**真的而没有定义，那么使用from sound.effects import 这种语法的时候，就不会*导入包:mod:sound.effects里的任何子模块。他只是把包:mod:sound.effects和它里面定义的所有内容导入进来（可能运行:file:**init**.py里定义的初始化代码）。

这会把 :file:**init**.py里面定义的所有名字导入进来。并且他不会破坏掉我们在这句话之前导入的所有明确指定的模块。看下这部分代码:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

这个例子中，在执行from...import前，包:mod:sound.effects中的echo和surround模块都被导入到当前的命名空间中了。（当然如果定义了**all**就更没问题了）

通常我们并不主张使用*这种方法来导入模块，因为这种方法经常会导致代码的可读性降低。不过这样倒的确是可以省去不少敲键的功夫，而且一些模块都设计成了只能通过特定的方法导入。

记住，使用from Package import specific_submodule这种方法永远不会有错。事实上，这也是推荐的方法。除非是你要导入的子模块有可能和其他包的子模块重名。

如果在结构中包是一个子包（比如这个例子中对于包:mod:sound来说），而你又想导入兄弟包（同级别的包）你就得使用导入绝对的路径来导入。比如，如果模块:mod:sound.filters.vocoder 要使用包:mod:sound.effects中的模块:mod:echo，你就要写成 from sound.effects import echo。

```
from . import echo
from .. import formats
from ..filters import equalizer
```

无论是隐式的还是显式的相对导入都是从当前模块开始的。主模块的名字永远是"**main**"，一个Python应用程序的主模块，应当总是使用绝对路径引用。

包还提供一个额外的属性，:attr:**path**。这是一个目录列表，里面每一个包含的目录都有为这个包服务的:file:**init**.py，你得在其他:file:**init**.py被执行前定义哦。可以修改这个变量，用来影响包含在包里面的模块和子包。

这个功能并不常用，一般用来扩展包里面的模块。

Python 输入和输出

在前面几个章节中，我们其实已经接触了 Python 的输入输出的功能。本章节我们将具体介绍 Python 的输入输出。

输出格式美化

Python两种输出值的方式: 表达式语句 和 print() 函数。(第三种方式是使用文件对象的 write() 方法; 标准输出文件可以用 sys.stdout 引用。)

如果你希望输出的形式更加多样，可以使用 str.format() 函数来格式化输出值。

如果你希望将输出的值转成字符串，可以使用 repr() 或 str() 函数来实现。

str() 函数返回一个用户易读的表达形式。

repr() 产生一个解释器易读的表达形式。

例如

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) +
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

这里有两种方式输出一个平方与立方的表:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line 注意前一行 'end' 的使用
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

注意：在第一个例子中, 每列间的空格由 print() 添加。

这个例子展示了字符串对象的 rjust() 方法, 它可以将字符串靠右, 并在左边填充空格。

还有类似的方法, 如 ljust() 和 center()。 这些方法并不会写任何东西, 它们仅仅返回新的字符串。

另一个方法 zfill(), 它会在数字的左边填充 0，如下所示：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

str.format() 的基本使用如下:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

括号及其里面的字符 (称作格式化字段) 将会被 format() 中的参数替换。

在括号中的数字用于指向传入对象在 format() 中的位置，如下所示：

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

如果在 format() 中使用了关键字参数, 那么它们的值会指向使用该名字的参数。

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

位置及关键字参数可以任意的结合:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Mar
                                                        other='Georg
The story of Bill, Manfred, and Georg.
```

'!a' (使用 ascii()), '!s' (使用 str()) 和 '!r' (使用 repr()) 可以用于在格式化某个值之前对其进行转化:

```
>>> import math
>>> print('The value of PI is approximately {}.'.format(math.pi))
The value of PI is approximately 3.14159265359.
>>> print('The value of PI is approximately {!r}.'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

可选项 ':' 和格式标识符可以跟着字段名。 这就允许对值进行更好的格式化。 下面的例子将 Pi 保留到小数点后三位：

```
>>> import math
>>> print('The value of PI is approximately {0:.3f}.'.format(math.p
The value of PI is approximately 3.142.
```

在 ':' 后传入一个整数, 可以保证该域至少有这么多的宽度。 用于美化表格时很有用。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

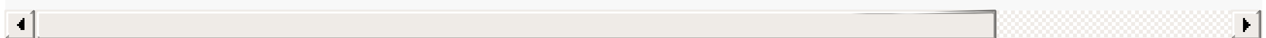
如果你有一个很长的格式化字符串, 而你不想将它们分开, 那么在格式化时通过变量名而非位置会是很好的事情。

最简单的就是传入一个字典, 然后使用方括号 '[]' 来访问键值 :

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
          'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

也可以通过在 table 变量前使用 '**' 来实现相同的功能 :

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.forr
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```



旧式字符串格式化

% 操作符也可以实现字符串格式化。 它将左边的参数作为类似 sprintf() 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串. 例如:

```
>>> import math
>>> print('The value of PI is approximately %5.3f.' % math.pi)
The value of PI is approximately 3.142.
```

因为 str.format() 比较新的函数， 大多数的 Python 代码仍然使用 % 操作符。但是因为这种旧式的格式化最终会从该语言中移除, 应该更多的使用 str.format().

读和写文件

open() 将会返回一个 file 对象，基本语法格式如下:

```
open(filename, mode)
```

实例:

```
>>> f = open('/tmp/workfile', 'w')
```

- 第一个参数为要打开的文件名。
- 第二个参数描述文件如何使用的字符。 mode 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写。 mode 参数是可选的; 'r' 将是默认值。

文件对象的方法

本节中剩下的例子假设已经创建了一个称为 f 的文件对象。

f.read()

为了读取一个文件的内容，调用 f.read(size), 这将读取一定数目的数据, 然后作为字符串或字节对象返回。

size 是一个可选的数字类型的参数。 当 size 被忽略了或者为负, 那么该文件的所有内容都将被读取并且返回。

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline()

f.readline() 会从文件中读取单独的一行。换行符为 '\n'。f.readline() 如果返回一个空字符串, 说明已经已经读取到最后一行。

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

f.readlines()

f.readlines() 将返回该文件中包含的所有行。

如果设置可选参数 sizehint, 则读取指定长度的字节, 并且将这些字节按行分割。

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\
```

另一种方式是迭代一个文件对象然后读取每行:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

这个方法很简单, 但是并没有提供一个很好的控制。 因为两者的处理机制不同, 最好不要混用。

f.write()

f.write(string) 将 string 写入到文件中, 然后返回写入的字符数。

```
>>> f.write('This is a test\n')
15
```

如果要写入一些不是字符串的东西, 那么将需要先进行转换:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
18
```

f.tell()

f.tell() 返回文件对象当前所处的位置, 它是从文件开头开始算起的字节数。

f.seek()

如果要改变文件当前的位置, 可以使用 f.seek(offset, from_what)函数， from_what 表示开始读取的位置，offset表示从from_what再移动一定量的距离，比如 f.seek(10, 3)表示定位到第三个字符并再后移10个字符。

from_what值为0时表示文件的开始，它也可以省略，缺省是0即文件开头。下面给出一个完整的例子：

```
>>> f = open('/tmp/workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # 移动到文件的第六个字节
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # 移动到文件的倒数第三字节
13
>>> f.read(1)
b'd'
```

f.close()

在文本文件中 (那些打开文件的模式下没有 b 的), 只会相对于文件起始位置进行定位。

当你处理完一个文件后, 调用 f.close() 来关闭文件并释放系统的资源，如果尝试再调用该文件，则会抛出异常。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
<pre>
<p>
当处理一个文件对象时, 使用 with 关键字是非常好的方式。在结束后, 它会帮你正确的
<pre>
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

文件对象还有其他方法, 如 isatty() 和 trucate(), 但这些通常比较少用。

pickle 模块

python的pickle模块实现了基本的数据序列和反序列化。

通过pickle模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。

通过pickle模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

基本接口：

```
pickle.dump(obj, file, [,protocol])
```

有了 pickle 这个对象, 就能对 file 以读取的形式打开:

```
x = pickle.load(file)
```

注解：从 file 中读取一个字符串，并将它重构为原来的python对象。

file: 类文件对象，有read()和readline()接口。

实例1：

```
#使用pickle模块将数据对象保存到文件

import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

实例2：

```
#使用pickle模块从文件中重构python对象

import pprint, pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

Python 错误和异常

作为Python初学者，在刚学习Python编程时，经常会看到一些报错信息，在前面我们没有提及，这章节我们会专门介绍。

Python有两种错误很容易辨认：语法错误和异常。

语法错误

Python 的语法错误或者称之为解析错，是初学者经常碰到的，如下实例

```
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

这个例子中，函数 print() 被检查到有错误，是它前面缺少了一个冒号（:）。

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的箭头。

异常

即便Python程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

异常以不同的类型出现，这些类型都作为信息的一部分打印出来: 例子中的类型有 ZeroDivisionError，NameError 和 TypeError。

错误信息的前面部分显示了异常发生的上下文，并以调用栈的形式显示具体信息。

异常处理

以下例子中，让用户输入一个合法的整数，但是允许用户中断这个程序（使用 Control-C 或者操作系统提供的方法）。用户中断的信息会引发一个 KeyboardInterrupt 异常。

```
>>> while True:
        try:
            x = int(input("Please enter a number: "))
            break
        except ValueError:
            print("Oops!  That was no valid number.  Try again   ")
```

try语句按照如下方式工作；

- 首先，执行try子句（在关键字try和关键字except之间的语句）
- 如果没有异常发生，忽略except子句，try子句执行后结束。
- 如果在执行try子句的过程中发生了异常，那么try子句余下的部分将被忽略。如果异常的类型和 except 之后的名称相符，那么对应的except子句将被执行。最后执行 try 语句之后的代码。
- 如果一个异常没有与任何的except匹配，那么这个异常将会传递给上层的try中。

一个 try 语句可能包含多个except子句，分别来处理不同的特定的异常。最多只有一个分支会被执行。

处理程序将只针对对应的try子句中的异常进行处理，而不是其他的 try 的处理程序中的异常。

一个except子句可以同时处理多个异常，这些异常将被放在一个括号里成为一个元组，例如:

```
    except (RuntimeError, TypeError, NameError):
        pass
```

最后一个except子句可以忽略异常的名称，它将被当作通配符使用。你可以使用这种方法打印一个错误信息，然后再次把异常抛出。

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

try except 语句还有一个可选的else子句，如果使用这个子句，那么必须放在所有的except子句之后。这个子句将在try子句没有发生任何异常的时候执行。例如:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

使用 else 子句比把所有的语句都放在 try 子句里面要好，这样可以避免一些意想不到的、而except又没有捕获的异常。

异常处理并不仅仅处理那些直接发生在try子句中的异常，而且还能处理子句中调用的函数（甚至间接调用的函数）里抛出的异常。例如:

```
>>> def this_fails():
        x = 1/0

>>> try:
        this_fails()
    except ZeroDivisionError as err:
        print('Handling run-time error:', err)

Handling run-time error: int division or modulo by zero
```

抛出异常

Python 使用 raise 语句抛出一个指定的异常。例如:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

raise 唯一的一个参数指定了要被抛出的异常。它必须是一个异常的实例或者是异常的类（也就是 Exception 的子类）。

如果你只想知道这是否抛出了一个异常，并不想去处理它，那么一个简单的 raise 语句就可以再次把它抛出。

```
>>> try:
        raise NameError('HiThere')
    except NameError:
        print('An exception flew by!')
        raise

An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

用户自定义异常

你可以通过创建一个新的exception类来拥有自己的异常。异常应该继承自 Exception 类，或者直接继承，或者间接继承，例如:

```
>>> class MyError(Exception):
        def __init__(self, value):
            self.value = value
        def __str__(self):
            return repr(self.value)

>>> try:
        raise MyError(2*2)
    except MyError as e:
        print('My exception occurred, value:', e.value)

My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在这个例子中，类 Exception 默认的 **init**() 被覆盖。

当创建一个模块有可能抛出多种不同的异常时，一种通常的做法是为这个包建立一个基础异常类，然后基于这个基础类为不同的错误情况创建不同的子类:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is no
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

大多数的异常的名字都以"Error"结尾，就跟标准的异常命名一样。

定义清理行为

try 语句还有另外一个可选的子句，它定义了无论在任何情况下都会执行的清理行为。 例如:

```
>>> try:
        raise KeyboardInterrupt
    finally:
        print('Goodbye, world!')

Goodbye, world!
KeyboardInterrupt
```

以上例子洪不管try子句里面有没有发生异常，finally子句都会执行。

如果一个异常在 try 子句里（或者在 except 和 else 子句里）被抛出，而又没有任何的 except 把它截住，那么这个异常会在 finally 子句执行后再次被抛出。

下面是一个更加复杂的例子（在同一个 try 语句里包含 except 和 finally 子句）:

```
>>> def divide(x, y):
        try:
            result = x / y
        except ZeroDivisionError:
            print("division by zero!")
        else:
            print("result is", result)
        finally:
            print("executing finally clause")

>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

预定义的清理行为

一些对象定义了标准的清理行为，无论系统是否成功的使用了它，一旦不需要它了，那么这个标准的清理行为就会执行。

这面这个例子展示了尝试打开一个文件，然后把内容打印到屏幕上:

```
for line in open("myfile.txt"):
    print(line, end="")
```

以上这段代码的问题是，当执行完毕后，文件会保持打开状态，并没有被关闭。

关键词 with 语句就可以保证诸如文件之类的对象在使用完之后一定会正确的执行他的清理方法:

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

以上这段代码执行完毕后，就算在处理过程中出问题了，文件 f 总是会关闭。

Python 类

和其它编程语言相比，Python 在尽可能不增加新的语法和语义的情况下加入了类机制。

Python中的类提供了面向对象编程的所有基本功能：类的继承机制允许多个基类，派生类可以覆盖基类中的任何方法，方法中可以调用基类中的同名方法。

对象可以包含任意数量和类型的数据。

类定义

语法格式如下：

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

类实例化后，可以使用其属性，实际上，创建一个类之后，可以通过类名访问其属性。

类对象

类对象支持两种操作：属性引用和实例化。

属性引用使用和 Python 中所有的属性引用一样的标准语法：obj.name。

类对象创建后，类命名空间中所有的命名都是有效属性名。所以如果类定义是这样:

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

实例化类：

```
x = MyClass()
```

以上创建了一个新的类实例并将该对象赋给局部变量 x，x 为空的对象。

很多类都倾向于将对象创建为有初始状态的。因此类可能会定义一个名为 **init**() 的特殊方法（构造方法），像下面这样：

```
def __init__(self):
    self.data = []
```

类定义了 **init**() 方法的话，类的实例化操作会自动调用 **init**() 方法。所以在下例中，可以这样创建一个新的实例:

```
x = MyClass()
```

当然， **init**() 方法可以有参数，参数通过 **init**() 传递到类的实例化操作上。例如:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

类的方法

在类地内部，使用def关键字可以为类定义一个方法，与一般函数定义不同，类方法必须包含参数self,且为第一个参数:

```
#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
        self.__weight = w
    def speak(self):
        print("%s is speaking: I am %d years old" %(self.name,self.

p = people('tom',10,30)
p.speak()
```

继承

Python 同样支持类的继承，如果一种语言不支持继承就，类就没有什么意义。派生类的定义如下所示:

```
class DerivedClassName(BaseClassName1):
    <statement-1>
    .
    .
    .
    <statement-N>
```

需要注意圆括号中基类的顺序，若是基类中有相同的方法名，而在子类使用时未指定，python从左至右搜索 即方法在子类中未找到时，从左到右查找基类中是否包含方法。

BaseClassName（示例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用:

```
class DerivedClassName(modname.BaseClassName):
```

实例

```
#单继承示例
class student(people):
    grade = ''
    def __init__(self,n,a,w,g):
        #调用父类的构函
        people.__init__(self,n,a,w)
        self.grade = g
    #覆写父类的方法
    def speak(self):
        print("%s is speaking: I am %d years old,and I am in grade

s = student('ken',20,60,3)
s.speak()
```

多重继承

Python同样有限的支持多继承形式。多继承的类定义形如下例:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

需要注意圆括号中父类的顺序，若是父类中有相同的方法名，而在子类使用时未指定，python从左至右搜索 即方法在子类中未找到时，从左到右查找父类中是否包含方法。

```
#另一个类，多重继承之前的准备
class speaker():
    topic = ''
    name = ''
    def __init__(self,n,t):
        self.name = n
        self.topic = t
    def speak(self):
        print("I am %s,I am a speaker!My topic is %s"%(self.name,se

#多重继承
class sample(speaker,student):
    a =''
    def __init__(self,n,a,w,g,t):
        student.__init__(self,n,a,w,g)
        speaker.__init__(self,n,t)

test = sample("Tim",25,80,4,"Python")
test.speak()#方法名同，默认调用的是在括号中排前地父类的方法
```

类私有方法

__private_method 两个下划线开头，声明该方法为私有方法，不能在类地外部调用。

在类的内部调用slef.__private_methods。

类的专有方法：

- **init** 构造函数，在生成对象时调用
- **del** 析构函数，释放对象时使用
- **repr** 打印，转换
- **setitem**按照索引赋值
- **getitem**按照索引获取值
- **len**获得长度
- **cmp**比较运算
- **call**函数调用
- **add**加运算
- **sub**减运算
- **mul**乘运算
- **div**除运算
- **mod**求余运算
- **pow**称方

更多介绍请查看：http://www.w3cschool.cc/python/python-object.html

Python 标准库概览

操作系统接口

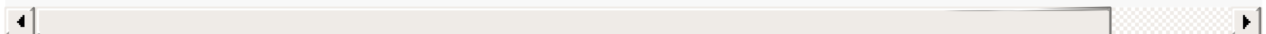
os模块提供了不少与操作系统相关联的函数。

```
>>> import os
>>> os.getcwd()      # 返回当前的工作目录
'C:\\Python34'
>>> os.chdir('/server/accesslogs')   # 修改当前的工作目录
>>> os.system('mkdir today')   # 执行系统命令 mkdir
0
```

建议使用 "import os" 风格而非 "from os import *"。这样可以保证随操作系统不同而有所变化的 os.open() 不会覆盖内置函数 open()。

在使用 os 这样的大型模块时内置的 dir() 和 help() 函数非常有用:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstri
```



针对日常的文件和目录管理任务，:mod:shutil 模块提供了一个易于使用的高级接口:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

文件通配符

glob模块提供了一个函数用于从目录通配符搜索中生成文件列表:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

命令行参数

通用工具脚本经常调用命令行参数。这些命令行参数以链表形式存储于 sys 模块的 argv 变量。例如在命令行中执行 "python demo.py one two three" 后可以得到以下输出结果:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

错误输出重定向和程序终止

sys 还有 stdin，stdout 和 stderr 属性，即使在 stdout 被重定向时，后者也可以用于显示警告和错误信息。

```
>>> sys.stderr.write('Warning, log file not found starting a new on
Warning, log file not found starting a new one
```

大多脚本的定向终止都使用 "sys.exit()"。

字符串正则匹配

re模块为高级字符串处理提供了正则表达式工具。对于复杂的匹配和处理，正则表达式提供了简洁、优化的解决方案:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

如果只需要简单的功能，应该首先考虑字符串方法，因为它们非常简单，易于阅读和调试:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

数学

math模块为浮点运算提供了对底层C函数库的访问:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random提供了生成随机数的工具。

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

访问 互联网

有几个模块用于访问互联网以及处理网络通信协议。其中最简单的两个是用于处理从 urls 接收的数据的 urllib.request 以及用于发送电子邮件的 smtplib:

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.p
...     line = line.decode('utf-8')  # Decoding the binary data to
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

注意第二个例子需要本地有一个在运行的邮件服务器。

日期和时间

datetime模块为日期和时间处理同时提供了简单和复杂的方法。

支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。

该模块还支持时区处理:

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03\. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

数据压缩

以下模块直接支持通用的数据打包和压缩格式：zlib，gzip，bz2，zipfile，以及tarfile。

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

性能度量

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python 提供了一个度量工具，为这些问题提供了直接答案。

例如，使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人的多,timeit 证明了现代的方法更快一些。

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

相对于 timeit 的细粒度，:mod:profile 和 pstats 模块提供了针对更大代码块的时间度量工具。

测试模块

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试

doctest模块提供了一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。

测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。

通过用户提供的例子，它强化了文档，允许 doctest 模块确认代码的结果是否与文档一致:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # 自动验证嵌入测试
```

unittest模块不像 doctest模块那么容易使用，不过它可以在一个独立的文件里提供一个更全面的测试集:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

免责声明

W3School提供的内容仅用于培训。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关。W3School简体中文版的所有内容仅供测试，对任何法律问题及风险不承担任何责任。